INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

N° 1248

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

PROOF METHODS OF DECLARATIVE PROPERTIES OF DEFINITE PROGRAMS

Pierre DERANSART

Juin 1990

RR_1248

PROOF METHODS OF DECLARATIVE PROPERTIES OF DEFINITE PROGRAMS

MÉTHODES DE PREUVE DE PROPRIÉTÉS DÉCLARATIVES DE PROGRAMMES DÉFINIS

Programme 1

Pierre DERANSART
INRIA
Domaine de Voluceau - Rocquencourt
B.P. 105
78153 LE CHESNAY Cedex
Tél. : (33-1) 39 63 55 36
email : deransar@minos.inria.fr

Abstract
In this paper we shall consider proofs of declarative properties of Definite Programs, i.e. properties associated with their logical semantics, in particular what is called the partial correctness of a logic program with respect to a specification and properties related to the proof theoretic semantics based on the notion of proof-tree. A specification consists of a logical formula associated with each predicate and establishing a relation between its arguments. A definite program is partially correct iff every possible answer substitution satisfies the specification.
This paper generalizes known results in logic programming in three ways : first it considers any kind of specification, second its results can be applied to extensions of logic programming such as functions or constraints, third new proof methods (the annotations methods) are presented. It gives an unified framework for different kind of known proof methods like the consequence verification method, the structural induction in definite clauses or the inductive proof of formulas.
Two proof methods are presented. The second is adapted from the Attribute Grammar field to the field of Logic Programming. Both are proven sound and complete. The first one consists of defining a specification stronger than the original one, which furthermore is inductive (fixpoint induction). Many aspects of the inductive specification method are investigated.The second method is a refinement of the first one : with every predicate, one associates a finite set of formulas (we call this an annotation), together with implications between formulas. The proofs become more modular and tractable, but the user has to verify the consistency of his proof, which is a decidable property. This method is particularly suitable for proving the validity of specifications which are not inductive. It has the same power as the first method, but it is more specifically adapted to prove properties holding inside the proof-trees.

Keywords : Logic Programming, Proof methods, Fixpoint Induction, Annotations, Declarative properties.

Résumé
Dans cette étude, nous abordons le problème des preuves de Propriétés Déclaratives de Programmes Définis. Les propriétés déclaratives sont celles qui sont associées à la sémantique logique. Nous voulons donc traiter de ce que l'on appelle habituellement la correction partielle d'un programme logique relativement à une spécification et de propriétés reliées à la sémantique basée sur la notion d'arbre de preuve. Une spécification consiste en des formules logiques associées à chaque prédicat et définissant une relation entre ses arguments. Un programme défini est partiellement correct si et seulement si toutes les substitutions réponses possibles satisfont la spécification.
Ici, nous généralisons les résultats connus dans ce domaine dans trois directions : d'une part, il n'y a aucune restriction sur le type de spécification considéré, d'autre part, les résultats obtenus pour les programmes définis peuvent s'étendre à des extensions de la programmation en logique comme des fonctions ou des contraintes, enfin une nouvelle famille de méthodes de preuve est présentée : la méthode des annotations. Nous donnons une présentation unifiée des différentes méthodes connues comme la "consequence verification method", la récurrence structurale et les preuves de formules par récurrence.
Deux méthodes sont étudiées. La seconde résulte d'une adaptation de résultats obtenues dans le domaine des grammaires attribuées. On montre ici que les deux méthodes sont fiables et on étudie leur complétude. La première consiste à définir une spécification plus forte que l'originale et à prouver qu'elle est récurrente (récurrence du point fixe). On analyse plusiseurs aspects de cette méthode basée sur la récurrence. La seconde méthode peut-être vue comme un raffinement de la première : au lieu d'une seule formule on associe plusieurs formules en nombre fini à un prédicat (on appelle ceci une annotation), mais on indique en complément comment les petites formules dépendent les unes des autres dans les clauses. On ajoute ainsi un facteur de modularité supplémentaire à la preuve (déjà structurée par la clause) et la rend plus aisée. En revanche, il devient nécessaire de s'assurer de sa cohérence qui est en fait une propriété décidable. Cette méthode est particulièrement bien adaptée pour prouver des spécifications non récurrentes ; elle n'ajoute cependant pas de puissance à la première. Plus important est le fait qu'elle est surtout utile pour prouver des propriétés des arbres de preuve eux-mêmes.

Mots clés : Programmation en Logique, Méthodes de Preuves, Récurrence du point fixe, Annotations, Propriétés déclaratives.

PROOF METHODS OF DECLARATIVE PROPERTIES OF DEFINITE PROGRAMS

Introduction

The problem of proving the partial correctness of a definite program (in which the clauses have exactly one positive literal), with respect to a given specification, saying what the answer substitutions (if any) should be has been considered many times [Cla 79, Hog 84, Dev 87, DrM87, SS 86, BC 89] but rarely studied in deep. The purpose of this paper is to present an unified view of the partial correctness proof methods of logic programs based on the notion of proof of properties of the proof-trees.

It is largely accepted that definite programs have an implicit actual semantics (the program itself or its completion [Llo 87] or the set of its atomic logical consequences), which is assimilated to its intended semantics. For definite programs all the formalizations of the actual semantics coincide and we will consider that it is the set of all the (not necessarily ground) proof-trees. It is known that all the proof-tree roots correspond to the atomic logical consequences of the program and conversely. Unfortunately the actual and the intended semantics (i.e. the intention of the programmer) do not coincide necessarily. The practice of software development and the relatively low level of expression in logic programming imposes to give attention to validation methods. These are based on the comparison of the actual semantics of a definite program and its intended semantics.

A program is partially correct w.r.t. its intended semantics if the relation it specifies corresponds to the intended semantics. By "intended semantics" we mean some property expressed in some logical language. Depending on the degree of refinement of its expression, one may be interested to prove a weaker or a stronger property.

For example considering the classical definite program :

plus(zero, X, X)
plus(s(X), Y, s(Z)) ← plus(X, Y, Z)

The property saying that "all the elements of the denotation have the form plus(s^n(zero), t, s^n(t)) for $n \geq 0$ and t beeing any term" is the strongest possible one. But one could be interested to prove only that "the first argument of **plus** has always the form s^n(zero)" or "if the second argument is ground then the third is too and conversely". These last propositions are a kind of partial specification or partial intended semantics. An other kind of "partial specification" which is considered in the litterature [Kan 86] is to express some intended property of the program like the commutativity of "plus" with the given axioms interpreted on natural integers. This can be expressed by the formula :

$$\forall X, Y, Z \text{ integer}(X) \wedge \text{integer}(Y) \Rightarrow (\text{plus}(X, Y, Z) \Rightarrow \text{plus}(Y, X, Z))$$

We will show that all these properties can be proven using the same kind of induction on the structure of the program.

But our purpose is to do more : we introduce a new method to prove properties holding anywhere inside the proof-trees : the annotation method. By this method it is possible to prove also properties of the prof-tree roots hence properties of the actual semantics. Moreover it is possible to improve considerably the inductive methods by introducing more modularity : in the inductive methods a big assertion is proven holding at the roots of the proof-trees. In the method by annotation littler assertions will be used, instead of a big one, flowing though the proof-tree. It is still a "syntax directed" proof method but with "modular assertions". It will be shown that the assertions used in an inductive proof method may have an exponential size w.r.t. all the assertions used in an annotation method.

Furthermore the annotation proof-method is a solution for an other kind of modularity which can be formalized by a "typing" approach. In practice one is not interested in proving properties holding in all the prof-trees but only in some of them, i.e. for a subset of the prof-trees. For example the program "plus" given above, even if it is part of a larger program which uses other function symbols, will be ordinarily used assuming that all arguments are just integers. This hypothesis is not implicit by the given axioms of "plus". In particular, in a context with lists, the atom "plus (zero, [], [])" is a logical consequence of the axioms. But under the hypothesis that "one of the second or the third argument of the root is an integer" any proof-tree will be completely instanciated by integers. This can be formalized by an annotation, i.e. a set of formulas associated to the predicates. For example, with "plus", three formulas of the form "the i^{th} argument is an integer" and to which directions are assigned like : the second formula is inherited (hypotheses issued from the upper context), the others are synthesized (conclusions holding at the roots by assuming the hypotheses). Now we will say that a definite program is correct w.r.t. a given annotation (or the annotation is valid for the program), if in all the proof-trees whose root satisfies its hypothesis (inherited assertions) then all the assertions (inherited and synthesized) hold anywhere inside the proof-tree (and of course at the root). The program "plus" is correct w.r.t. the given annotation ; that is to say, if the second argument of the root is an integer then the whole proof-tree "contains" integers only.

Let us explain briefly how such approach may reduce the conceptual complexity of a proof. Assume that one wants to prove that the program "plus" specifies an addition on natural integers. It seems natural to try to prove its correctness w.r.t. the specification associated to the relation "plus(X, Y, Z)" : $Z =_{int} X +_{int} Y$
i.e. the intended semantics is expressed as a formula

$$Z = X + Y$$

in which the symbols "=" and "+" are respectively relational and functional symbols whose interpretation is defined for natural integers only, but not for other values (there is no interpretation for $[] +_{int} []$ for example). In practice partial functions and predicates are used.

If one assumes that the domains of interpretation are natural integers only, the proof by induction of such a formula is very easy to perform. This is the way such a program is usually understood when writing and reading its axioms. But finally an other property has been proven which is :

$$integer(X) \wedge integer(Y) \wedge integer(Z) \Rightarrow Z =_{int} X +_{int} Y$$

because by doing a simple proof by induction with partial functions, one make implicit hypotheses about the types of the arguments. But as we have seen it is possible to perform separately the proof concerning the type of the arguments and the correctness assertion "Z = X + Y".

These observations basically motivate the choices made in the presentation of this paper : a many sorted approach is considered allowing the handle assertions with partial functions in a very modular way. Here, we don't want just to obtain theoretical results but also to show how proofs can be organized such that they become simpler to handle and more tractable on big programs.

We achieve this introduction by a short presentation of the proof methods. Basically this paper has two parts corresponding to the two proof methods : the proof by induction and the proof by annotations. Both are syntax directed but, although the first can be viewed as a particular case of the second, it is studied separately. The reason comes from the fact that on one side most of the known literature is devoted to the first one and on the other side most of the basic theoretical results are obtained with it. The first method is devoted to the proof of partial correctness (validity of a specification) when the second is devoted to the proof of properties of the proof-trees.

Let us state more formally the definition of partial correctness.

A specification consists of a logical formula associated with each predicate of a definite program P, and establishing a relation between its arguments. If **S** is the family of such logical formulas :

$$\mathbf{S} = \{ S^p \mid \text{for each predicate in P} \}$$

and $\mathbb{D}$ the interpretation of the logical language including the functional symbols of P, then the partial correctness of P with regards to **S** is expressed as follows :

$$\forall p \text{ predicate of P}, \forall T \text{ list of term arguments if } P \models pT \text{ then } \mathbb{D} \models S^p(T)$$

what formally expresses that every atomic logical consequence of the program P (the actual semantics) satisfies the specification **S**.

The first method consists of defining a specification stronger than the original one, which furthermore is inductive (fixpoint induction). A specification is inductive if the axioms of the program are still valid in $\mathbb{D}$ when one replaces in the clauses the predicate occurrences by their specification. Many consequences and applications of the fixpoint view will be explored; in particular those following from the induction principle contained in the definite clauses of a logic program.

Our second method is a generalization of the first one : with every predicate we associate a finite set of formulas (we call this an annotation), together with implications between formulas in the clauses. The proofs become more modular and tractable, but their consistency has to be proven ; this is a decidable property. This method is particularly suitable for proving the validity of specifications which are not inductive, or to make type verifications inside the proof-trees.

Let us consider the previous example with one predicate more whose intended semantics is to specify the addition in natural integers and the length of a list :

```
plus(zero, X, X)        ←
plus(s (X), Y, s (Z))   ←   plus(X, Y, Z).
plength([ ], zero)      ←
plength([A|L], N)       ←   plength(L, M), plus(s (zero), M, N).
```

and the following specification : $\mathbf{S} = \{ S^{plus}, S^{length} \}$

$S^{plus}(X, Y, Z) : Z = X + Y$
$S^{length}(L, N) : N = \text{length}(L)$

assuming that X, Y, Z, N are integers and L a list.

We first prove the validity of the specification **S** for the given program using the first method, then we prove with the annotation method that the proof-trees are well-typed.

This first specification is inductive (hence the program is correct w.r.t. **S**). In fact the axioms can rewritten :

$X = 0 + X$
$Z = X + Y \Rightarrow Z + 1 = X + 1 + Y$
$\text{length}([ ]) = 0$
$\text{length}(L) = M \text{ and } N = M + 1 \Rightarrow \text{length}([A|L]) = N$

which are valid formulas in the domains of the natural integers (X, Y, Z, M, N), lists (L) and unspecified elements (A).

Note that partial correctness does not say anything about the existence of solutions (completeness) or effective form of the goals during or after computations ("run-time properties" in [DrM 87], "STP" in [DF 88]), or whether any solution can be reached using some computation rule (termination, [FGK 85]). All together these problems are part of the total correctness of a definite clause program. Partial correctness states only that any computed atom — if any — (which belongs necessarily to the actual semantics) satisfies the specification.

Now we use the annotation method to prove the well-typing. We consider the following annotation :

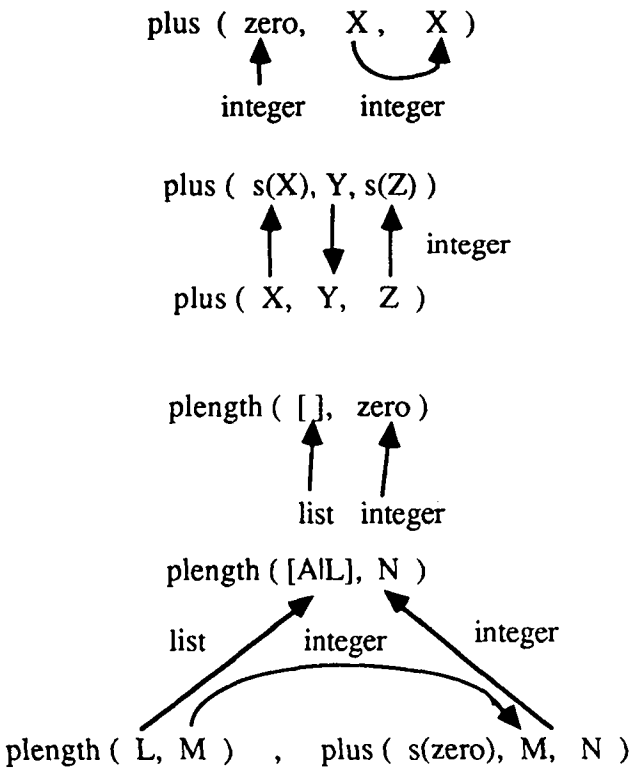
Inherited assertion for plus : integer(second argument)

Synthesized assertion for plus : integer(first argument), integer(third argument).

No inherited assertion for plength.

Synthesized assertion for plength : list(first argument), integer(second argument).

The proof is performed showing locally in each clause how the type informations are propagated. The following figure illustrates the way to propagate the information "integer" or "list" inside the clauses :



As there is no possible cycle following the arrows in any proof-tree the annotation is valid, that is to say that the proof-trees of roots "plus" whose second argument is an integer are of type integer and the proof-trees of root "plength" are well-typed (there is no inherited hypothesis attached to "plength").

Notice that our definition of the specifications, as our example as well, do not restrict the domains of the interpretations to be Herbrand bases (we call them term bases). This will permit to establish

general results holding not only for pure logic programming but also for extensions like functional or constraint logic programming.

The paper is organized as follows :

Section 1 introduces the basic definitions and notations. Special emphasis is given to the term (or Herbrand) interpretations which corresponds to the most usual way to deal with the semantics of logic programs.

Section 2 introduces the definite clause programs, their semantics and the specifications. Two notions of validity are defined in order to deal with general interpretations, not with Herbrand's ones only. This will permit to obtain very general results on the proof methods.

Section 3 presents the inductive proof method and explores its properties. Results of completeness, relative completeness (in the sense of Cook [Coo 78]) and incompleteness are established. Relationships with fixpoint and structural induction are investigated as the extension of the method to general programs and other kind of properties.

Section 4 presents the proof method with annotations which can be viewed as a generalization of the inductive method but is not more powerful (It does not permit to prove more valid specifications). In contrast it permits to prove properties holding inside proof-trees.

In *section 5* a (short) example is fully developed with the purpose to show the originality and the usefulness of the annotation method.

Finally a comparison with other results published in the literature is provided in *section 6*.

1 - Basic definitions and notations

(1.1) Sort, signatures, terms.

Context free grammars and logical languages will be defined in the algebraic style of [CD 88]. Some definitions are also useful to describe logic programs.

Let S be a finite set of sorts. A S-sorted signature F is a finite set of function symbols with two mappings : the domain α (some word in S* representing the sorts of the arguments in the same order), the sort σ of the function symbol. The length of $\alpha(f)$ is called the arity of f and denoted $\rho(f)$. If $\alpha(f) = \varepsilon$ (the empty word) then f is a constant symbol. The pair $< \alpha(f), \sigma(f) >$ is the profile of f. A constant of sort s has profile $< \varepsilon, s >$.

A heterogeneous F-algebra is an object $\mathbb{A}$:

$$\mathbb{A} = < \{A_S\}_{S \in S}, \{f_A\}_{f \in F} >$$

where $\{A_S\}$ is a family of non empty sets indexed by S (the carriers) and each f_A a mapping :

$$A_{S1} \times \ldots \times A_{Sn} \rightarrow A_S \text{ if f has profile } < s1 \ldots sn, s >.$$

Let V be a S-sorted set of variables (each v in V has arity ε, sort $\sigma(v)$ in S). The free F-algebra $\mathbb{T}$ generated by V, also denoted T(F, V) is identified as usual as the set of the well-formed terms, "well typed" with respect to sorts and arities. Terms will also be identified with trees in a well

known manner. T(F) denotes the set of all terms without variables, i.e. the ground terms, $T(F)_S$ denotes the set of the ground terms of sort s.

A term t in $T(F)_S$ is considered as denoting a value t_A in A_S for a F-algebra A. For any F-algebra A and a S-sorted set of variables, an assignment of values in A_S to variables V_S, for all s in S, is an S-indexed family of functions.

$$\nu = \{ \nu_S : V_S \rightarrow A_S \}_{S \in S}$$

It is well-known that this assignment can be extended into a unique homomorphism

$$\nu' = \{ \nu'_S : T(F, V)_S \rightarrow A_S \}_{S \in S}$$

In $\mathbb{T}$ assignments are called substitutions. For any assignment ν in $\mathbb{T}$ and term t in T(F, V), νt is called an instance of t.

(1.2) Grammars

Proof-trees of a logic program (see definition 2.5) can be thought of as abstract syntax trees with associated atoms. These abstract syntax trees can be represented by abstract context free grammars. An abstract context free grammar is the pair < N, P > where N is a finite set (the non-terminal alphabet) and P a N-sorted signature (for more details see [DM 85]).

(1.3) Many sorted logical languages

The specifications will be given in some logical language together with an interpretation that we define as follows. Let S be a finite set of sorts containing the sort bool of the boolean values true, false. Let V be a sorted set of variables, F a S-signature and R a finite set of many sorted relation symbols (i.e. a set of symbols, each of them having an arity and, implicitely, the sort bool).

A logical language L over V, F, R is the set of formulas written with V, F, R and logical connectives like $\forall$, $\exists$, $\Rightarrow$, $\wedge$, $\vee$, not, ... We denote by free (φ) the possibly empty set of the free variables of the formula φ of L (free (φ) $\subseteq$ V), by $\mathbb{AND}$ A (resp $\mathbb{OR}$ A) the conjunction (resp. the disjunction) of formulas ($\mathbb{AND}$ $\emptyset$ = true, $\mathbb{OR}$ $\emptyset$ = false), and by φ $[u_1/v_1,..., u_n/v_n]$ the result of the substitution of u_i for each free occurrence of v_i, or $\varphi[u_1, \ldots, u_n]$. We do not restrict a priori the logical language to be first order.

Let $\mathbf{C}(L)$ denote a class of L-structures or L-interpretations, i.e. objects of the form :

$$\mathbb{D} = < \{D_S\}_{S \in S} , \{f_D\}_{f \in F}, \{r_D\}_{r \in R} >$$

where $< \{D_S\}_{S \in S} , \{f_D\}_{f \in F} >$ is a heterogeneous F-algebra and for each r in R, r_D is a total mapping

$$D_{s1} \times \ldots \times D_{sn} \rightarrow \{\text{true}, \text{false}\} = \text{bool} \quad \text{if } \alpha(r) = s1 \ldots sn.$$

The notion of validity is defined in the usual way. For every assignment ν, every $\mathbb{D}$ in $\mathbf{C}(L)$, every φ in L, one assumes that $(\mathbb{D}, \nu) \models \varphi$ either holds or does not hold. We say φ is valid in $\mathbb{D}$, and write $\mathbb{D} \models \varphi$, iff $(\mathbb{D}, \nu) \models \varphi$ for every assignment ν. In the sequel we will make a large use of a distinction which is usual in the field of logic programming [Llo 87]. We will distinguish the

algebraic part of a L-interpretation, called a L-preinterpretation (L may be omitted if there is no ambiguity). It is the interpretation of the function symbols of L. Let us call J a (L-) preinterpretation. Then $\mathbb{D}$ is a J based L-interpretation iff $\mathbb{D}$ is the L-interpretation consisting of J and the interpretation of the relations R of L.

(1.4) Examples

Here are some examples of interpretations for some languages :

Integers language

S = { int, pint, bool }

L is V variables

F = { zero, s, p, +}

R = { plus, = }

Interpretation $\mathbb{N}$ (natural integers)

Domains :	N_{int}	=	nat, nat = posnat U {0}
	N_{pint}	=	posnat (posnat = {1, 2, ...})
	N_{bool}	=	bool
Functions :	$zero_N$	$\in$	$<\varepsilon$, nat> : 0
	s_N	$\in$	<nat, posnat> : $n \rightarrow n + 1$
	p_N	$\in$	<posnat, nat> : $n \rightarrow n - 1$
	$+_N$	$\in$	<nat nat, nat> : $(n, m) \rightarrow n + m$
Relations :	$=_N$	$\in$	<nat, nat, bool> : $(n, m) \rightarrow$ true iff $n = m$
	$plus_N$	$\in$	<nat nat nat, bool> : $(n, m, l) \rightarrow$ true iff $l = n + m$

Interpretation $\mathbb{E}$ (non standard signed integers) [Bid 81]

Domains :	E_{int}	=	$\{(n, s) \mid n \in nat, s \in \{\pm 1\}\}$
	E_{posint}	=	$\{(n, s) \mid n \in posnat, s \in \{\pm 1\}\}$
	E_{bool}	=	bool
Functions :	$zero_E$	$\in$	$<\varepsilon, E_{int}>$: (0, + 1)
	s_E	$\in$	$<E_{int}, E_{posint}>$: $(n, s) \rightarrow (n + 1, s)$
	p_E	$\in$	$<E_{posint}, E_{int}>$: $(n, s) \rightarrow (n - 1, s)$
	$+_E$	$\in$	$<E_{int}\ E_{int}, E_{int}>$: $(n, s_1)\ (m, s_2) \rightarrow (n + m, s_2)$
Relations :	$=_E$	$\in$	$<E_{int}\ E_{int}$, bool> : $(n, s_1)\ (m, s_2) \rightarrow$ true iff $n=m$ and $s_1 = s_2$
	$plus_E$	$\in$	$<E_{int}\ E_{int}\ E_{int}$, bool> : $plus((n, s_1), (m, s_2), (l, s_3)) \rightarrow$ true iff $l = n + m$ and $s_3 = s_2$

Note that $\mathbb{N}$ as $\mathbb{E}$ are models of the axioms :

(1) plus(zero, X, X)

(2) plus(sX, Y, sZ) $\Leftarrow$ plus(X, Y, Z)

i.e. that these formulas are valid in $\mathbb{N}$ and $\mathbb{E}$.

But the formula :

(3) plus(X, Y, Z) $\Leftarrow$ plus(Y, X, Z). (commutativity)

is valid in $\mathbb{N}$ but not in $\mathbb{E}$ (the commutativity is not a logical consequence of axioms (1) and (2) ; i.e. it does not hold in all models of (1) and (2)).

Algebraic language

Given a set of sorts S increased with {bool, int}and a language L over V, F, R as in 1.1 in which F contains also {size} and R {ground, =, var}.

Interpretation $\mathbb{T}$ (canonical term interpretation)

Domains :

$T(F, V)_s$ set of the well formed terms of sort s following sorts and arities, built as follows :
if $1 \leq i \leq n$, $t_i \in T(F, V)_{si}$ and $f \in \langle s_1 \ldots s_n, s\rangle$
then $f(t_1, \ldots, t_n) \in T(F, V)_s$

$T_{bool} = \underline{bool}$

$T_{int} = N_{int}$

Functions $f_T \in \langle T(F, V)_{s1} \ldots T(F, V)_{sn}, T(F, V)_s \rangle$: f iff $f \in \langle s_1 \ldots s_n, s\rangle$

$size_T \in \langle T(F), T_{int}\rangle$: is defined recursively as follows :

$size_T(f) = 1$ if f has arity 0 (constant)

and $size_T(f(t_1, \ldots, t_n)) = 1 + \sum_{1 \leq i \leq n} size_T(ti)$.

Relations

$ground_T \in \langle T(F, V), \underline{bool}\rangle$: $ground_T(t) = \underline{true}$ iff t is a term without variable.

$=_T \in \langle T(F, V)\ T(F, V), \underline{bool}\rangle$: $t_1 =_T t_2$ iff t_1 and t_2 are syntactically the same (or t_1 and t_2 are formally equal).

$var_T \in \langle T(F, V), \underline{bool}\rangle$: $var_T(t) = \underline{true}$ iff $t \in V$.

Notice that $ground_T$ and $size_T$ are polymorphic and that $size_T$ is a partial function on T (F, V).

$=_T$, $=_E$ and $=_N$ satisfy the congruence axioms of the equality as given in [Llo 87]. For example the formula $X_1 =_E Y_1 \wedge X_2 =_E Y_2 \Rightarrow X_1 +_E X_2 =_E Y_1 +_E Y_2$ is valid in $\mathbb{E}$.

The following formula is valid in $\mathbb{T}$.

$X =_T Y \Rightarrow size_T(X) =_N size_T(Y)$

If there is no ambiguity following the typing convention of the operators, subscripts denoting the models will be omitted.

Lists language

S = {list, d-list, any, int, T, bool}

F = { [], [_|_], nil, ., append, -, repr, length }

[] and nil are constants, repr has arity 1, the others arity 2.

R = { is-a-list, is-a-dlist, permut, = }

Interpretation $\mathbb{L}$ (lists and difference-lists)

Domains :

L_{list} = lists as usual (defined by 'nil' and '.' denoting 'cons')

$L_{d\text{-}list}$ = d-lists (difference-lists as usual [Tär 86])

L_{any} = elem (type of the lists elements : no restriction)

L_{int} = nat (as in $\mathbb{N}$)

L_T = is T(F, V) as in $\mathbb{T}$

L_{bool} = bool

Functions :

$[]_L \in <\varepsilon, lists>$
$nil_L \in <\varepsilon, lists>$ | both denote the empty list, nil.

$[\_|\_]_L \in <elem\ lists, lists>$
$._L \in <elem\ lists, lists>$ | both denote the usual list constructor cons

$append_L \in <elem\ lists, lists>$: concatenation of lists. "$append_L$" operator will be omitted in long formulas if there is no ambiguity on the type of the arguments.

$-_L \in <lists\ lists, d\text{-}lists>$: difference-lists constructor

$repr_L \in <d\text{-}lists, lists>$: defines the list represented by a difference-list

$length_L \in <lists, nat>$: the length of a list

Relations :

is-a-list $\in$ <T (F, V), bool> : is-a-list(l) is true iff l is a list built with list constructors

is-a-d_list $\in$ <T (F, V), bool> : is-a-d_list(dl) is true iff all instances of dl are d_list built with d_list constructors and represent a list (i.e. $repr_L$ (dl) is defined).

For example is-a-d_list([X, Y | L] - [Y | L]) is true but is-a-d_list([X, Y | L] - [Z | L]) is false.

permut $\in$ <lists lists, bool> = permut(l_1, l_2) is true iff l_2 is a permutation of l_1

$=_L \in <lists\ lists, bool>$: $l_1 =_L l_2$ is true iff l_1 and l_2 are element by element identical lists (using $=_{elem}$).

(1.5) Term interpretations

A special attention should be given to the term-interpretations. They play a central role in logic programming as they are used to define the proof-theoretic and the fixpoint semantics of logic programs and as such they are also related to the computability of the relations defined by a logic program.

The basic idea of a term interpretation leads in its term domains : every value is denoted by a specific term (different from the others). Usually term interpretations are one-sorted but they can be many sorted as in (1.4 - algebraic language).

The term domain is T(F, V). It is closed by substitution. If V is empty (no variable) the term domain is ground and is also called the Herbrand universe. A term base B (or Herbrand base if there is no variable) is the set of atoms r (t_1, ..., t_n) with r in R and the t_i's in T(F, V). For more details on term bases see [Fer 85].

A term interpretation is defined by the canonical term preinterpretation in which the symbols are interpreted by themselve and for each r in R, by a subset of B defining r.

By definition of a term base, defining a term interpretation $\mathbb{T}$ is just to define $r_{\mathbb{T}}$ for every r in R. Then one can identify the assignment in $\mathbb{T}$ with the substitutions ranging over terms and, moreover, for any assignment v, the value of a term by v is vt (v applied to t) and the notion of validity : ($\mathbb{T}$, v) |= r (t_1, ..., t_n) is equivalent to (vt_1, ..., vt_n) $\in$ $r_{\mathbb{T}}$.

A term model of a set of formulas S is a term interpretation which satisfies all the formulas in S. A ground term model is also called a Herbrand model.

In an interpretation $\mathbb{D}$ the values of the domains are assumed different. Thus in a term interpretation all terms which are not formally equal are different. This can be axiomatized by the following axioms which we will denote as the term-equality axioms (EQ). We borrow them from [Apt 87] :

1. forall (x = x).

2. forall($f(x_1, ..., x_n) = f(y_1, ..., y_n)$ if $x_1 = y_1$ and ... $x_n = y_n$) for all function symbols f.

3. forall((p(y_1, ..., y_n) if p(x_1, ..., x_n)) if $x_1 = y_1$ and ... $x_n = y_n$) for all predicate p including "="

4. forall($x_1 = y_1$ and ... $x_n = y_n$ if $f(x_1, ..., x_n) = f(y_1, ..., y_n)$) for all function symbols f.

5. forall($f(x_1, ..., x_n \neq g(y_1, ..., y_m)$) for all pairs of different function symbols.

6. forall(t[x] $\neq$ x) for all terms t[x] containing the variable x and different from x.

Note that in every interpretation $\mathbb{D}$ containing an interpretation of the equality which satisfies these axioms, if a value is represented by a term, this term is unique. In other terms, this means that all constants and functions are considered as constructors of different values, hence there is no axiom on constructors (initial algebra approach). This point will not be developed any more here.

2 - Definite Clauses Programs, specifications

(2.1) Definition : Definite Clause Program (DCP).

A DCP is a triple P = <PRED, FUNC, CLAUS> where PRED is a finite set of predicate symbols, FUNC a finite set of function symbols disjoint of PRED, CLAUS a finite set of clauses defined as usual [Cla 79, Llo 87] with PRED and TERM = T(FUNC, V). Complete syntax can be seen in examples (2.2) and (2.3). A clause is called a fact if it is restricted to an atomic formula. Clauses are built as usual with PRED and TERM.

(2.2) Example : Program "plus"

PRED = {plus} ρ(plus) = 3
FUNC = {zero, s} ρ(zero) = 0, ρ(s) = 1
CLAUS = {c1 : plus (zero, X, X) ← ,
c2 : plus (s(X), Y, s(Z)) ← plus (X, Y, Z)}

variables begin with uppercase letters.

(2.3) Example : Program "permutations"

"List" terms are represented in Edinburgh syntax

PRED = {perm, extract} ρ(perm) = 2, ρ(extract) = 3
FUNC = {[],[_|_]} ρ([]) = 0, ρ([_|_]) = 2
CLAUS = {c1 : perm ([], []) ← ,
c2 : perm ([A|L], [B|M]) ← perm(N, M), extract([A|L], B, N) ,
c3 : extract ([A|L], A, L) ← ,
c4 : extract ([A|L], B, [A|M]) ← extract (L, B, M)}

(2.4) Definition : denotation of a DCP P : DEN(P)

The denotation of a DCP is the set of all its (not necessarily ground) atomic logical consequences :

$$DEN(P) = \{a \mid P \vdash a\}$$

We do not give any more details on the notions of models of P (structures in which the clauses are valid formulas) and of logical consequences (all atoms of DEN(P) are valid in the models of P), since we won't make use in this paper of the logical semantics of a logic program, but rather of its proof theoretic semantics that we shall now define. Other details can be found in [Cla 79, AvE 82, Llo 87, Fer 85].

(2.5) Definition : J-based proof-tree, proof-tree

Given a DCP P = < PRED, FUNC, CLAUS > and a pre-interpretation J s.t. J interprets all FUNC in P then the set of J-based proof-trees of P is defined as follows :

1 - If A is the result of the interpretation in J of the arguments of a fact in CLAUS whose variables (if any) have been assigned to some value, then the tree consisting of one vertice with label A is a J-based proof-tree.

2 - If $T_1,..., T_q$ for some $q > 0$ are J-based proof-trees with roots labelled $B_1,..., B_q$ and if $A \leftarrow B_1,..., B_q$ is the result of the interpretation in J of all the arguments of the atoms of some clause in CLAUS whose variables (if any) have been assigned to some value, then the tree consisting of the root labelled with A and the subtrees $T_1,..., T_q$ is a J-based proof-tree.

Intuitively, if we call *J-based instances of a clause* the result of assigning values to the variables of a clause and interpreting in J the term arguments, it is easy to observe that a proof-tree is built by pasting together J-based instances of clauses, such that the leaves are J-based instances of facts. If this last condition is removed one gets the notion of *partial proof-tree*.

Given an interpretation $\mathbb{D}$ of a class of L-structures such that L includes FUNC but not PRED, we denote by $PTR_{\mathbb{D}}(P)$ the set of all the proof-tree roots (root labels) of the J-based proof-trees of P, in

which J is the algebraic part of $\mathbb{D}$. If there is no confusion we will speak of the $\mathbb{D}$-based proof-trees instead of J-based.

If J is a term preinterpretation then one finds the usual notion of (not necessarily ground) proof-trees [Cla 79, Fer 85, DM 85]. Note that every instance of a proof-tree is a proof-tree also. The name "proof-tree" will be reserved to denote term-based proof-trees. In this case $PTR_T(P)$ is just denoted PRT(P).

(2.6) Proposition [Cla 79] **- Proof theoretic semantics.**
Given a DCP P : DEN(P) = PTR(P).

Thus, instead of the logical semantics of a logic program, one can deal with its proof-theoretic semantics. As pointed out in [DM 85, DM 88], proof-trees can be thought as syntax trees (terms of a clauses-algebra) "decorated" by atoms as specified in the proof-tree definition. Thus inductive proof methods as defined in [CD 88] may be applied to logic programs. This will be done in the section 3. Note that the set of all the $\mathbb{D}$-based instances of the elements of DEN(P) — let us denote it $DEN(P)_{\mathbb{D}}$ — does not correspond in general to $PTR_{\mathbb{D}}(P)$, but is included in. They are the same if $\mathbb{D}$ is a term interpretation or if in $\mathbb{D}$ all values are represented by a term and all the axioms of EQ are valid. The consequence of the difference in the general case will be studied in section (3.11).

It may be useful to observe that $PTR_{\mathbb{D}}(P)$ defines with $\mathbb{D}$ an interpretation which is a model of P. In particular, if $\mathbb{D}$ is a term-based interpretation, DEN(P) defines a model of P [Cla 79].

(2.7) Definition : Specification of a logic program.

A specification of a logic program P is a family of formulas $\mathbf{S} = \{ S^p \}_{p \in PRED}$ of a logical language L over V, F, R such that V contains the variables used in P and F contains FUNC, together with a L-structure $\mathbb{D}$. For every p of PRED, we denote by $\underline{varg}(p) = \{ p1, ..., p\rho(p) \}$ the set of variable names denoting any possible term in place of the 1^{st}, ... or $\rho(p)^{th}$ argument of p. Thus we impose $\underline{free}(S^p) \subseteq \underline{varg}(p)$.

For a good understanding of the results presented in this paper, it is important to remark that the relations R in the language L may or may not include PRED. There are two cases. L may contain PRED and a specification **S** may use theses predicates. Moreover the family of formulas may be reduced to the form $S^p : p(p_1, ..., p_n)$. In this case $\mathbb{D}$ can be viewed as a preinterpretation for P augmented with the interpretations of the relations in PRED. In the second case — R does not contain PRED — then $\mathbb{D}$ is an interpretation of L and can simply be viewed as a preinterpretation for P. Without lost of generality unless explicitely stated, it will be assumed in the sequel that the specification language L does not contain the predicate symbols of P. A specification **S** may also be viewed as defining an interpretation of the predicates in PRED as : $p(v_1, ..., v_n) \in \mathbb{D}$ iff $(\mathbb{D}, (v_1, ..., v_n)) \models S^p$. We will denote $\mathbf{I_S}$ the induced interpretation defined by a specification S ($\mathbf{I_S}$ is the union of the $\mathbf{I_S^p}$ for every p in PRED) and $\mathbf{S_{P,\mathbb{D}}}$ the family of formulas such that $\mathbf{I}_{S_{P,\mathbb{D}}} = PTR_{\mathbb{D}}(P)$. One will assume that such formulas always exist (this will be discussed in section 3.10). Given $\mathbb{D}$ a preinterpretation for P and the formulas **S**, $\mathbb{D}$ augmented by $\mathbf{I_S}$ will be denoted $\mathbb{D}_S$. $\mathbb{D}_S$ is a model of P if in particular $\mathbf{S} = \mathbf{S_{P,\mathbb{D}}}$. $\mathbb{D}_S$ with $\mathbf{S} = \mathbf{S_{P,\mathbb{D}}}$ will be denoted $\mathbb{D}_P$.

(2.8) Definition : valid specification.

A specification $\mathbf{S}$ on (L, $\mathbb{D}$) is valid for the DCP P (or P is correct w.r.t. $\mathbf{S}$) iff

$\forall p(u_1, ..., u_n) \in PTR_{\mathbb{D}}(P), \quad \mathbb{D} \models \mathbf{S}^p[u_1/p_1, ..., u_n/p_n], \quad n = \rho(p).$

In practice there is a more interesting definition of a "valid specification" which we will call "computational correctness" if we need to distinguish it from the previous one.

A specification P on (L, $\mathbb{D}$) is computationally valid (or P is computationally correct w.r.t. $\mathbf{S}$) iff

$\forall\ p(t_1, ..., t_n) \in DEN(P) \quad \mathbb{D} \models \mathbf{S}^p[t_1, ..., t_n]$

This last definition has been used in [Der 89]. Without the precautions taken here, the completeness of the method stated in theorem (3.5) does not hold. Nevertheless with any kind of interpretation $\mathbb{D}$ if a specification is valid in the sense of the $\mathbb{D}$-based proof-trees, it is also in the sense of DEN as $DEN(P)_{\mathbb{D}} \subseteq PTR_{\mathbb{D}}(P)$.

This second definition means that every atom of the denotation satisfies the specification (with a universal quantification of the variables in the terms), hence every atom in any proof-tree. It means also that every answer substitution (if any) satisfies the specification.

Both correspond to a notion of partial correctness referring to the declarative (i.e. proof-theoretic or logical) semantics since nothing is specified about the existence of proof-trees (the denotation can be empty). The way to compute them or the kind of resulting answer substitution for a given goal. If $DEN(P)_{\mathbb{D}}$ and $PTR_{\mathbb{D}}(P)$ are the same then both definitions coincide. Until section (3.14) the first definition will be used only.

Notice that any logical consequence of a valid specification is also valid, i.e. if $\mathbf{S}$ is valid and $D \models \mathbf{S} \Rightarrow \mathbf{S'}$ then $\mathbf{S'}$ is valid. ($\mathbf{S} \Rightarrow \mathbf{S'}$ stands as a shorthand for the family of implications $\mathbf{S}^p \Rightarrow \mathbf{S'}^p$ for all p in PRED).

(2.9) Example : specification for (2.2)

L_1 = V_1 contains varg(plus) = {plus1, plus2, plus3}
F_1 = {zero, s, + }
R_1 = { = }

$\mathbb{D}_1$ = $\mathbb{N}$ as in (1.4)

$\mathbf{S}_1$ = { $\mathbf{S}_1^{plus}$ }, $\mathbf{S}_1^{plus}$: plus3 = plus1 + plus2

The validity of $\mathbf{S}_1$ (which is proven in the next section) means that the program "plus" in (2.2) specifies the addition in $\mathbb{N}$, in particular that every n-uple of values corresponding to the interpreted arguments of the elements of the denotation satisfies the specification plus3 = plus1+ plus2 .

More precisely, it means that if the variables appearing in a proof-tree are assigned over the domains of the functions or predicate in which they appear (i.e. here N_{int}), all the atoms at the nodes of the proof-tree satisfy the corresponding specification if function symbols are interpreted as in $\mathbb{N}$. If one

wants to make clear that this specification holds for integer values only (if for example this program may be used in different contexts), one could use the following specification :

$$S_{11}^{plus} \quad = \quad (integer(plus2) \vee integer(plus3)) \Rightarrow plus3 = plus1 + plus2$$

In this case it will be clear that plus1, plus2 and plus3 are always integers as the following specification is also valid :

$$S_{12}^{plus} \quad : \quad integer(plus1) \wedge (integer(plus2) \Leftrightarrow integer(plus3))$$

An other property may be interesting :

L_2 = V_2 as in L_1 , F_2 = { zero, s}
R_2 = { ground } ρ(ground) = 1.

$\mathbb{D}_2$ = $\mathbb{T}$ as in (1.4)

S_2 = { S_2^{plus} }, S_2^{plus} : (ground(plus3) $\Leftrightarrow$ ground(plus2)) $\wedge$ ground(plus1)

S_2 is a valid specification (it can be observed on every proof-tree and will be proven in the next section).

(2.10) Example : specification for (2.3).

This example uses a many sorted L_3 structure.

L_3 = V_3 contains varg(perm) and varg(extract).
F_3 = { [], [_|_], nil, . , append } [] and nil are constants,
the other operators have arity 2.
R3 = { is-a-list, permut } ρ(is-a-list) = 1, ρ(permut) = 2.
$\mathbb{D}_3$ = $\mathbb{L}$ as in (1.4)
S_3 = { S^{perm} : permut (perm1, perm2) ,
$S^{extract}$: $\exists$ L1, L2 (extract1 = append (L1, extract2.L2)
$\wedge$ extract3 = append (L1, L2) }

3 - Inductive proof method

(3.1) Definition : Inductive specification S of a DCP P.

A specification S on (L, $\mathbb{D}$) of a DCP P is inductive iff for every c in CLAUS,

$$c: r_0(t0_1, ..., t0n_0) \Leftarrow r_1(t1_1, ..., t1n_1), ..., r_m(tm_1, ..., tmn_m) ,$$
$$\mathbb{D} \models (\underset{1 \le k \le m}{AND} S^{rk} [tk_1, ..., tkn_k] \Rightarrow S^{r0} [t0_1, ..., t0n_0]) \qquad (1)$$

i.e. a specification is inductive iff in every clause, if the specification holds for the atoms of the body, it holds for the head. Remark that in (1) the remaining variables are variables of the clause c and they are universally quantified.

(3.2) Proposition : an inductive specification is valid.

If $\mathbf{S}$ of P is inductive then $\mathbf{S}$ is valid for P .

Proof : by an easy induction on the size of $\mathbb{D}$-based the proof-trees, if $PTR_{\mathbb{D},n}(P)$ denotes the set of all the roots of the proof-tree of size $\leq n$, $\mathbf{S}$ holds in $PTR_{\mathbb{D},n+1}(P)$ (by definitions (2.5) and (3.1) and the notion of validity), thus in $PTR_{\mathbb{D}}(P) = \bigcup_{n \geq 0} PTR_{\mathbb{D},n}(P)$. QED.

(3.3) Definition : stronger (weaker) specification.

Let $\mathbf{S}$ and $\mathbf{S}'$ be two specifications of P on (L, $\mathbb{D}$). One says that $\mathbf{S}$ is weaker than $\mathbf{S}'$ (or $\mathbf{S}'$ stronger than $\mathbf{S}$), and denotes it by $\mathbb{D} \models (\mathbf{S}' \Rightarrow \mathbf{S})$ iff $\forall p \in PRED$, $\mathbb{D} \models (\mathbf{S}'^{p} \Rightarrow \mathbf{S}^{p})$.

We denote $\mathbf{S}_{true}$ the specification such that $\mathbf{S}^{p}_{true}$: true for all p in PRED (i.e no specification).

(3.4) Proposition : the strongest specification is inductive.

Given a DCP P, $\mathbf{S}_{P,\mathbb{D}}$ and $\mathbf{S}_{true}$ are respectively the strongest and the weakest valid specification for P and $\mathbf{S}_{P,\mathbb{D}}$ is inductive i.e. all valid specifications S satisfy :

$$\mathbb{D} \models (\mathbf{S}_{P,\mathbb{D}} \Rightarrow \mathbf{S} \Rightarrow \mathbf{S}_{true}), \quad \text{and } \mathbf{S}_{P,\mathbb{D}} \text{ is inductive.}$$

Proof : it is easy to observe that $\mathbf{S}_{P,\mathbb{D}}$ is inductive as it corresponds by definition to the interpretation defined by $PTR_{\mathbb{D}}(P)$ whose elements are $\mathbb{D}$-based proof-tree roots (i.e. built with $\mathbb{D}$-based instances of clauses). On the other hand every valid specification is by definition true for every n-uple of values of a $\mathbb{D}$-based proof-tree root. Hence $\mathbb{D} \models \mathbf{S}_{P,\mathbb{D}} \Rightarrow \mathbf{S}$. Obviously $\mathbb{D} \models \mathbf{S} \Rightarrow \mathbf{S}_{true}$. QED.

(3.5) Theorem (soundness and completeness of the inductive proof method)

A specification $\mathbf{S}$ on (L, $\mathbb{D}$) is valid for P iff it is weaker than some inductive specification $\mathbf{S}'$ on (L, $\mathbb{D}$) :

i.e. 1) $\exists\, \mathbf{S}'$ inductive
2) $\mathbb{D} \models \mathbf{S}' \Rightarrow \mathbf{S}$.

Proof : Soundness is trivial since if $\mathbf{S}'$ is inductive, it is valid by proposition (3.2) and if $\mathbb{D} \models \mathbf{S}' \Rightarrow \mathbf{S}$ then $\mathbf{S}$ is valid by the remark in (2.8).

Completeness results from proposition (3.4) with $\mathbf{S}' = \mathbf{S}_{P,\mathbb{D}}$. Notice that one does not have the completeness if one restricts the specification language to be first order (See (3.10)).

(3.6) Example : the specification $\mathbf{S}_1$ (2.9) is inductive.

Following the definition (3.1) it is sufficient to prove :

$$\mathbb{N} \models \mathbf{S}_1[\text{ zero }, X, X] \text{ and}$$
$$\mathbb{N} \models \mathbf{S}_1[X, Y, Z] \Rightarrow \mathbf{S}_1[s(X), Y, s(Y)]$$

or :

$\mathbb{N} \models 0 + X = X$ and

$\mathbb{N} \models (X + Y = Z \Rightarrow X+1 + Y = Z+1)$

which are valid formulas in $\mathbb{N}$.

(3.7) Example : the specification S_2 (2.9) is inductive.

In the same way it is easy to show that the following formulas are valid on $\mathbb{D}_2$:

ground(zero) $\wedge$ ground(X) $\Rightarrow$ ground(X)
and
[ground(Z) $\Rightarrow$ (ground(X) $\wedge$ ground(Y))] $\Rightarrow$ [ground(s(Z)) $\Rightarrow$ (ground(s(X)) $\wedge$ ground(Y)]

(3.8) Example : the specification S_3 (2.10) is inductive.

It is easy to show that the following formulas are valid on $\mathbb{D}_3$: (some replacements are already made in the formulas and universal quantifications on the variables are implicit)

in c1 : permut (nil, nil)

in c2 : permut (N, M) $\wedge$ $\exists$ L1, L2 (A.L = append(L1, B.L2) $\wedge$ N = append (L1, L2))
$\Rightarrow$ permut (A.L, B.M)

in c3 : $\exists$ L1, L2 (A.L = append(L1, A.L2) $\wedge$ L = append (L1, L2))
(L1 and L2 are lists) take L1 = nil and L2 = L.

in c4 : $\exists$ L1, L2 L = append(L1, B.L2) $\wedge$ M = append (L1, L2))
$\Rightarrow$ $\exists$ L'1, L'2 (A.L = append(L'1, B.L'2) $\wedge$ A.M = append (L'1, L'2))
take L'1 = A.L1 and L'2 = L2.

(3.9) More examples

We achieve this illustration with some more examples of inductive proofs.

CONCATENATION

We define the concatenation of difference lists as usual by one fact :

concat (L1 - L2, L2 - L3, L1 - L3) $\leftarrow$

Note that we introduce explicitely the typing predicates as "repr" is a partial function over difference lists.

$S^{concatenate}$: is-a-d_list (concatenate1) $\wedge$ is-a-d_list (concatenate2)
$\Rightarrow$ repr(concatenate3) = append(repr(concatenate1), repr(concatenate2))

defined on (L_3, $\mathbb{D}_3$) is inductive.

The claim is obvious, there is only one fact and :
is-a-d_list (L_1 - L_2) $\wedge$ is-a-d_list (L_2 - L_3)
$\Rightarrow$ repr(L1-L3) = append(repr(L1- L2), repr(L2 -L3))

GRAPH COLOURING

We consider a graph colouring program whose (second order) specification is the following :
$\exists$ f : regions $\rightarrow$ colours such that
$\forall r_i, r_j$ regions, $r_i \neq r_j \wedge$ adjacent(r_i, r_j) $\Rightarrow f(r_i) \neq f(r_j)$.

We denote by solution(f) the property required for the mapping f.

The interpretation we are interested in consists of a description of pairs of different adjacent regions and different non adjacent ones and of mappings which will be represented by lists of pairs <region, colour>.

Here is a definite program whose purpose is to specify mappings satisfying P_solution.

c1 : P_solution([]) $\leftarrow$

c2 : P_solution([<R, C>]) $\leftarrow$ colour(C), region(R).

c3 : P_solution([$<R_1, C_1>$, $<R_2, C_2>$ | S]) $\leftarrow$ P_adjacent(R_1, R_2),
diff_C(C_1, C_2),
P_solution([$<R_1, C_1>$ | S]),
P_solution([$<R_2, C_2>$ | S]).

c4 : P_solution([$<R_1, C_1>$, $<R_2, C_2>$ | S]) $\leftarrow$ P_noadjacent(R_1, R_2),
P_solution([$<R_1, C_1>$ | S]),
P_solution([$<R_2, C_2>$ | S]).

The following specification is inductive, hence valid :

S = { $\mathbf{S}^{solution}$: solution(solution1) $\wedge$ in solution1 all pairs are different (no redundancy),

$\mathbf{S}^{adjacent}$: adjacent1 $\neq$ adjacent2 $\wedge$ adjacent(adjacent1, adjacent2) $\wedge$ adjacent1 and adjacent2 are regions,

$\mathbf{S}^{noadjacent}$: noadjacent1 $\neq$ noadjacent2 $\wedge \sim$ adjacent(noadjacent1, noadjacent2) $\wedge$ noadjacent1 and noadjacent2 are regions,

$\mathbf{S}^{diff\_C}$: $diff\_C_1 \neq diff\_C_2 \wedge$ $diff\_C_1$ and $diff\_C_2$ are colours,

$\mathbf{S}^{colour}$: colour1 is a colour,

$\mathbf{S}^{region}$: region1 is a region}

One may assume that "P_adjacent", "P_noadjacent" and "diff_C" are given by facts satisfying the corresponding specification (adjacent arguments are regions and diff_C arguments are colours). It remains to prove that **S** is inductive in the clauses of "solution".

In c1 and c2 the result holds trivially.

In c3 : one may prove separately :

$$\text{solution}([\langle R_1, C_1\rangle \mid S]) \wedge \text{solution}([\langle R_2, C_2\rangle \mid S]) \wedge C_1 \neq C_2 \wedge R_1 \neq R_2 \wedge \text{adjacent}(R_1, R_2) \Rightarrow \text{solution}([\langle R_1, C_1\rangle, \langle R_2, C_2\rangle \mid S])$$

and

$$(\text{no redundancy in } [\langle R_1, C_1\rangle \mid S] \wedge \text{no redundancy in } [\langle R_2, C_2\rangle \mid S] \wedge C_1 \neq C_2 \wedge \text{adjacent}(R_1, R_2)) \wedge R_1 \neq R_2) \Rightarrow \text{no redundancy in } [\langle R_1, C_1\rangle, \langle R_2, C_2\rangle \mid S].$$

It is obvious.

In c4 : same kind of reasoning.

(3.10) Wand's incompleteness results

The theorem 3.5 which states the completeness of the method has been obtained assuming that the formulas $S_{P, \mathbb{D}}$ always exists. We now turn to the problem whether such formulas exists or not.

In [CD 88] it is shown that the Wand's incompleteness results established for Hoare's like deductive systems holds also for inductive proofs in attribute grammars, that is to say that the assertion language, if restricted to first order (finite) formulas, may be not rich enough to express the (inductive) properties needed to achieve the proof of some specification.

For this reason we did not restrict our specification languages to be first order and we pointed out (definition 2.7) the need of a language large enough such that the result on the completeness of the method could be stated (theorem (3.5)).

However one could suspect that such incompleteness results might not hold, as we are concerned by definite programs which are, as shown in [CD 88], very particular attribute grammars. In other words one could expect that starting from definite programs which are already first order logical specifications, the inductive proof method could still remain complete in the framework of first order logic.

Unfortunately it is not the case. In other words the formulas $S_{P, \mathbb{D}}$ on $(L, \mathbb{D})$ do not always exist if one restricts the language L to be first order. We give a formal proof of this results by re-coding the Wand's counter example [Wan 78, CD 88] in the definite program's style.

We show an example of definite program P and a valid specification for P defined on a L-structure $\mathbb{M}$ such that no first order inductive specification defined with L can be found.

Let L be the language defined as follows :

Let **p, q, r** be unary predicates and **f,** a unary function symbols. It is assumed a predicate "=" which will be interpreted as the identity of the values of the domains.

Let $\mathbb{M}$ be the first order L-structure such that its domain is M with :

$M = \{ a_n \mid n \geq 0 \} \cup \{ b_n \ (n \geq 0 \}$

$f \in < M, M>$ is $f(a_0) = a_0, f(b_0) = b_0$
$f(a_n) = a_{n-1}, f(b_n) = b_{n-1}$ for $n \geq 1$

$p(X)$ true iff $X = a_0$

$q(X)$ true iff $X = b_0$

$r(X)$ true iff $X = a_n$ for some n of the form $k(k + 1)/2$.

It is shown in Wand [Wan 78, theorem 2] that there is no first order formula φ with one free variable X such that for d in M, $(\mathbb{M}, d) \models \varphi$ iff $d \in \{ a_n \mid n \geq 0 \}$.

Let P be the following pure logic program :

PRED = { rec }

FUNC = { f }

CLAUS = { rec(X, Y) $\leftarrow$ rec(f(X), Y) ,
rec(X, X) $\leftarrow$ equal(f(X), X) ,
equal(X, X) $\leftarrow$ }

It is easy to see that the strongest specification $S_{P, \mathbb{M}}$ for P is the following (inductive, but we will show that S^{rec} is not first order expressible in L) :

S^{rec} : $(rec1 \in \{ an \mid n \geq 0 \} \wedge rec2 = a_0) \vee (rec1 \in \{b_n \mid n \geq 0 \} \wedge rec2 = b_0)$

S^{equal} : $equal1 = equal2$

Let Θ be the valid specification such that :

$\Theta^{rec} = r\ (rec1) \Rightarrow p\ (rec2)$

$\Theta^{X} = \underline{true}$ for $X \in \{ equal \}$

Assume there is an inductive specification Θ_i such that $\Theta_i \Rightarrow \Theta$. It must satisfy the following conditions :

(1) $\Theta_i^{rec}\ (rec1, rec2) \wedge r\ (rec1) \Rightarrow p\ (rec2)$

(2) $\Theta_i^{rec}\ (f(X)\ Y) \Rightarrow \Theta_i^{rec}\ (X, Y)$

(3) $f(X) = X \Rightarrow \Theta_i^{rec}\ (X, X)$ (Θ_i^{equal} is $equal1 =_M equal2$)

Let now φ be the formula :

$\varphi\ (x) : \forall\ y\ \Theta_i^{rec}\ (x, y) \Rightarrow p(y)$

We will show that $(\mathbb{M}, d) \models \varphi$ iff $d \in \{ a_n \mid n \geq 0 \}$.

if part :

if $\Theta_i^{rec}(a_n, v)$, $n \geq 0$ holds then by (2) $\Theta_i^{rec}(a_m, v)$, for all $m > n$. Let us choose $m > n$ such that $r(a_m)$ holds, thus by (1), $p(v)$ holds also.

only part :

from (3), $\Theta_i^{rec}(b_0, b_0)$ holds, hence by (2) $\Theta_i^{rec}(b_n, b_0)$ for $n \geq 0$. Now take $\Theta_i^{rec}(b_n, v)$ with $v = b_0$; it holds, but since $p(b_0)$ does not, $(\mathbb{M}, d) \models \varphi$ does not hold if $d \notin \{ a_n \mid n \geq 0 \}$.

By the Wand's result φ cannot be first order, hence Θ_i^{rec} cannot be too.

Remark that in this example we used a non deterministic coding of the deterministic one given in [Wan 78, CD 88].

(3.11) The relative completeness of the proof method holds also with term interpretations

The result of relative completeness has been obtained with an example whose logic program has an empty denotation. It corresponds to a limit case in which $DEN(P)_{\mathbb{D}}$ (here empty) and $PTR_{\mathbb{D}}(P)$ are different, as illustrated on Fig. 3.13.

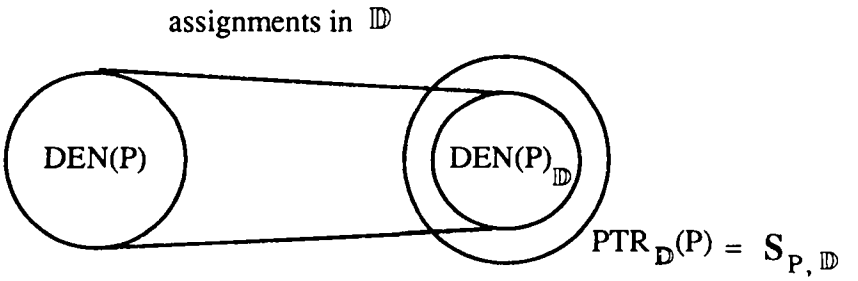


Figure 3.13

If $\mathbb{D}$ is a term interpretation $DEN(P)_{\mathbb{D}}$ and $PTR_{\mathbb{D}}(P)$ are the same.

In this case validity and computational validity coincide and every element of $PTR_{\mathbb{D}}(P)$ is the root of a proof-tree obtained with "instances" of clauses of P in $\mathbb{D}$. Assume that $\mathbb{D}$ satisfies the equality axioms (EQ) and that every value in $\mathbb{D}$ can be represented by a term. Then to every instance of a clause it corresponds an instance in the term-interpretation, hence the corresponding $\mathbb{D}$-based proof-tree roots are also elements of DEN(P). It follows that the theorem (3.5) gives a sound and complete method to prove the computational validity of specifications expressed on term interpretations.

By the proof above one could suspect that the incompleteness result originates in the possible differences between $DEN(P)_{\mathbb{D}}$ and $PTR_{\mathbb{D}}(P)$ due to unrestricted models, and that the method could remain complete with first order assertions in the case of term interpretations.

One could be confirmed in this opinion by the fact that in the Herbrand universe every value is uniquely described by a term and that all values are finitely represented. In the above example one could have represented all values of M by constants indexed in $\mathbb{N}$; doing that and with few modification of the logic program one becomes able to formulate the strongest inductive specification by a first order formula.

We show now that even with pure Herbrand domains the incompleteness result still holds. Let us consider the assertion $S_{P, \mathbb{D}}$ in the case of a term intepretation. Every formula $S^p{}_{P, \mathbb{D}}$ can be expressed as a (usually infinite) disjunction of equalities :

$$S^p{}_{P, \mathbb{D}} : \quad \bigvee_{p(\bar{t}) \in DEN(P)} \bar{p} = \bar{t}$$

Consider also a term interpretation in which the only predicate symbol is "=" which satisfies the axioms EQ. ("=" is interpreted as the term identity). The specification language L uses only the function symbols of the program P and the predicate symbol "=". Thus we are faced to the following problem : does there exists a first order formula equivalent to the (usually infinite) formula $S^p{}_{P, \mathbb{D}}$) ?

Assume there exists F^p equivalent to $S^p{}_{P, \mathbb{D}}$. $F^p[\bar{t}]$ is true iff $p[\bar{t}]$ belongs to DEN(P). It is shown in [CL 88] that there exists a decision procedure for the validity in the Herbrand universe of any first order formula with the only predicate symbol "=". It follows that if F^p exists for all predicates p of P, then DEN(P) is recursive (F decidable). But it is known that any Turing machine can be encoded by a definite program whose denotation contains only its halting states (see [DM 85] for such a description). Hence the incompleteness results.

The incompleteness of the method does not come from the nature of the interpretation but rather is relative to the nature of the language, too limited if restricted to the first order logic.

(3.12) Definitions : IF, ONLY-IF, IFF, COMP axioms

First of all let us recall some classical definitions borrowed from [Apt 87].

Given a DCP P, the following steps define the IF(P), ONLY-IF(P) and IFF(P).

step1 : remove terms
Transform each clause $p(t_1, \ldots, t_n) \leftarrow a_1, \ldots, a_m$ of P into $p(x_1, \ldots, x_n) \Leftarrow x_1 = t_1 \wedge \ldots \wedge x_n = t_n \wedge a_1 \wedge \ldots \wedge a_m$.

step2 : introduce existential quantifiers :
Let $y_1, \ldots, y_q$ be the variables of the original clause. Transform each formula $p(t_1, \ldots, t_n) \Leftarrow F$ into $p(t_1, \ldots, t_n) \Leftarrow \exists\, y_1, \ldots, y_q\ F$.

step3 : group similar formulas :
Let $p(x_1, \ldots, x_n) \Leftarrow F1, \ldots, p(x_1, \ldots, x_n) \Leftarrow Fk$ be all formulas obtained in the previous step with a relation p on the left hand side. Replace them by one formula : $p(x_1, \ldots, x_n) \Leftarrow F1 \vee \ldots \vee Fk$.

step4 : handel "undefined" relation symbols :

For each n-ary relation symbol q not appearing in a head of a clause in P add a formula : $q(x_1, \ldots, x_n) \Leftarrow \underline{\text{false}}$.

step5 : introduce universal quantifiers : IF axioms.
Replace each formulas F by $\forall$ (F).

step6 : add ONLY-IF axioms
For each formulas $\forall(a \Leftarrow F)$ obtained at step 5 add a new formula $\forall(F \Leftarrow a)$.

We call the intermediate form obtained after step5 the IF definition associated with P, denoted IF(P), the set of formulas added at step6 is the ONLY-IF definition associated with P, denoted ONLY-IF(P), and the union of both sets is the IFF definition, denoted IFF(P).

The completion of a program P is the union of IFF(P) and EQ (section 1.5). It is denoted COMP(P).

Given a set of formulas F we will denote by F[**S**] the replacement of all the occurrences of the atoms $p(t_1, \ldots, t_n)$ in F by $\mathbf{S}^p [t_1, \ldots, t_n]$, the formula associated to p in the specification **S** in which every free variable pi corresponding to the i^{th} argument is replaced by the term t_i appearing as the ith argument of the occurrence of p in F. It is important to observe that the models of IF(P) are the same than the models of P and that $\mathbb{D} \models P[\mathbf{S}]$ is equivalent to say that $I_{\mathbf{S}}$ defines, with $\mathbb{D}$ taken as a pre-interpretation for P, a model of P (i.e. $\mathbb{D}_{\mathbf{S}} \models P$).

(3.13) A view on fixpoint induction

In [Par 69] it is shown that the following proof rule (called fixpoint induction) is sound. We restate it in the logic programming framework :

(1) $\mathbb{D} \models IF(P)[\mathbf{S}]$ then (2) $\mathbb{D} \models Conv(\mathbf{P}) \Rightarrow \mathbf{S}$
where Conv(**P**) is the least fixpoint of the system (1) in the sense of Knaster-Tarski, i.e. the least solution **S** of (1) (which is also, by monotonicity, the least solution of IFF(P)[**S**] in $\mathbb{D}$).

By the same fixpoint theorem it is known that (3) $\mathbb{D} \models \mathbf{S}_{P, \mathbb{D}} \Rightarrow Conv(\mathbf{P})$ (this point will be clarified below). Hence the theorem 3.5 (soundness) is proven again as (1) means that **S** is inductive and from (2) and (3) **S** is valid ($\mathbb{D} \models \mathbf{S}_{P, \mathbb{D}} \Rightarrow \mathbf{S}$ is the definition 2.8 of validity).

This shows the way of some extensions of the proof method. The fixpoint induction is sound as far as the system IF(P)[**S**] is monotonic in **S** and easy to use if Conv(**P**) may be easily characterized in term of a valid specification.

The first point can be achieved if one considers logic programs built with clauses in which the body can be any first order formula, but in which the predicates to which a specification may be attached are positive. This includes bodies without negated predicates, or constraint logic programming in which the negations are in the constraints only, clearly distinguished from the other predicates. Thus as stated in [Par 69 (3.2)] if the body of the clauses involves just $\forall$, $\exists$, $\wedge$, ($\sim$ in constraints only), then the monotonicity is preserved.

As concerns the second point it is known that if P is a normal program (i.e. the bodies of the clauses may contain negated atoms) and if P is stratified in the sense of [Llo 87 pp 110] then there exists a strongest valid specification satisfying (1) expressed in term of a minimal model of the completion of P. But the system (1) is not always monotonic any more and (3) may not holf.

Joining both results one may expect to be able to use the same method for more general programs than definite ones. One extension could be logic programs without negation but any first order formula in the body. An other has been studied in [DF 89] for well founded logic programs.

An other but similar way to look at fixpoints is to consider the $\mathbb{D}$-based immediate consequence operator $T_{P,\mathbb{D}}$ as defined in [Llo 87, Apt 87] by :

$$T_{P,\mathbb{D}}(I) = \{ \quad va_0^{\mathbb{D}} \mid \exists\, C : a_0 \leftarrow a_1, \ldots, a_m \text{ and } \exists v \text{ s.t. } va_1^{\mathbb{D}} \in I, \quad i > 0 \quad \}$$

It is known that I is a model of IF(P) (or of P) iff $T_{P,\mathbb{D}}(I) \subseteq I$ and of ONLY-IF(P) iff $I \subseteq T_{P,\mathbb{D}}(I)$ and of IFF(P) iff $T_{P,\mathbb{D}}(I) = I$. Hence the fixpoint induction can be restated as follows :

(1)' $T_{P,\mathbb{D}}(I) \subseteq I$ then (2)' $LFP \subseteq I$

in which LFP is the least fixpoint of $T_{P,\mathbb{D}}$ which contains, by the monotonicity of $T_{P,\mathbb{D}}$, the set of the $\mathbb{D}$-based proof-tree roots, namely ($PTR_{\mathbb{D}}(P)$ is just $T_{P,\mathbb{D}} \uparrow \omega$) :

(3)' $PTR_{\mathbb{D}}(P) \subseteq LFP \subseteq I$. This gives an other formulation of the proof of theorem 3.5 (soundness).

(3.14) Proof method extends to amalgamation of DCP's and other programming

All the results expressed until this section do not require the interpretations to satisfy the properties of term domains (equality axioms). This means that the proof method holds for something obviously more general than just usual DCP's but also for DCP's and functional programming (or many kinds of amalgamations).

As an example consider the program FIB (Fibonnacci) :

FIB : fib (0, 1) ←
fib (1, 1) ←
fib (N, R1 + R2) ← N > 1, fib (N - 1, R1), fib (N - 2, R2)

in which all the function symbols are interpreted on $\mathbb{N}$ (1.4) (augmented with the predicate symbol > interpreted as usual) and the constants are the values of nat.

One can show by the inductive method that the following specification is inductive hence valid :

fib1 ∈ nat ∧ fib2 = fibonacci (fib1)

This example shows also the possibility to apply the same proof method to programs with built-in predicates (N > 1 in this example is interpreted as ">" in $\mathbb{N}$).

(3.15) An axiomatic view of the proof method

The proof method has been formulated until now by the way of interpretations. This is not the most usual approach especially if one wants to make automatized proofs. In this case it is assumed that a specification is expressed in a language whose meaning is given by a set of axioms, i.e. a subset of L denoted Ax.

The definitions of validity and inductive specification need to be adapted considering that it is a kind of generalization from one interpretation to a class of interpretations (those which satisfy the axioms Ax). These interpretations will be referred in the sequel as the interpretations of the axioms Ax. It should be clear that, due to the inclusion of all the function symbols of the programm into the language of the specifications, the interpretations which we will consider are also preinterpretations for P. It will be assumed also that there is no axiom concerning the predicates of P in Ax and that the interpretations do not include the predicates of P. If Ax contains an other axiomatisation of P most of the definitions are simplified and this will be discussed in the section 6.

(3.15.1) Definition : valid specification (axiomatic view)

A specification **S** on L with axioms Ax *is valid* for the DCP P (or P is correct w.r.t. **S**) iff

(val) for all p in PRED and every model $\mathbb{D}$ of Ax which is a preinterpretation for P,
if $p(\bar{v}) \in PTR_{\mathbb{D}}(P)$ then $\mathbb{D} \models \mathbf{S}^p[\bar{v}]$

(eval) A specification **S** on L with axioms Ax is **computationally valid** for P iff :
for all $p(\bar{t})$ in DEN(P) : $Ax \models \mathbf{S}^p[\bar{t}]$.

These definitions need some comments. First of all it is clear that (val) implies (eval), that is to say if a specification is valid it is computationally valid. This follows from $DEN(P)_{\mathbb{D}} \subseteq PTR_{\mathbb{D}}(P)$ for any interpretation $\mathbb{D}$. The converse does not hold. The definition 3.15.1 (eval) can be regarded as a generalization of the notion of computational validity to a class of models.

(3.15.2) Definition : inductive specification (axiomatic view)

A specification **S** on L with axioms Ax is inductive for the DCP P iff $Ax \models P[\mathbf{S}]$ (note that the definition 3.1 is $\mathbb{D} \models P[\mathbf{S}]$ with the notations of 3.11).

(3.15.3) Theorem (axiomatic view of theorem 3.5)

A specification **S** on L with axioms Ax is valid for P **if** it is weaker that some inductive specification **S'** on L, i.e. : there exists **S'** such that :

1) $Ax \models P[\mathbf{S'}]$ (**S'** inductive)

2) $Ax \models \mathbf{S'} \Rightarrow \mathbf{S}$ (for all p in PRED : $Ax \models \mathbf{S'}^p \Rightarrow \mathbf{S}^p$)

Proof : given $\mathbb{D}$ model of Ax, by hypothesis (1) $\mathbb{D} \models P[\mathbf{S'}]$, then **S'** is inductive, hence valid (def 2.8), i.e. $\forall\, p(\bar{v}) \in PTR_{\mathbb{D}}(P)$ then $\mathbb{D} \models \mathbf{S}^p[\bar{v}]$. QED.

Unfortunately the converse does not hold. (Demonstration in the annex).

Now it happens that the completion of a DCP is often regarded as a "specification" of its acutal semantics which can be used to prove the validity of a specification (hence its computational validity). The proof method is stated as follows :

(3.15.4) Proposition (Proof method with the completion)

A specification **S** on L with axioms Ax is valid for P if for all p in PRED ;

$$IFF(P) \cup Ax \models p(\bar{x}) \Rightarrow S^p[\bar{x}]$$

Proof : obvious as for any preinterpretation $\mathbb{D}$, model of Ax, $PTR_{\mathbb{D}}(P)$ satisfies $T_{P,\mathbb{D}}(PTR_{\mathbb{D}}(P)) = PTR_{\mathbb{D}}(P)$ and is a model of IFF(P). QED.

But the converse does not hold (incompleteness of the method). Both methods (3.15.3 and 3.15.4) are uncomparable (see demontration in the annex), but the inductive one is obviously more modular and thus more tractable. However both can be combined as it will be illustrated in the next section.

(3.16) Proving properties of the denotation

The inductive proof method can be used to prove properties of the least term models of a DCP P, DEN(P). In fact if $\mathbb{D}$ is the term interpretation defined by DEN(P) then the definition 2.8 becomes (validity and computational validity coincide) :

(1) $DEN(P) \models \overset{AND}{\underset{p}{}} \forall\ (p(\bar{x}) \Rightarrow S^p[\bar{x}])$

which is a consequence of **S** inductive, i.e. :

(2) $DEN(P) \models P[S]$.

This property may be used to prove the validity of formulas of the form (1) by "execution" because the validity of a formula F in a term model of P can be deduced from the following statments :

Statment1 : if a formula F is atomic, it is valid iff it belongs to DEN(P). Hence $\forall(F)$ $(\exists(F))$ is semi decidable by running the goal F in which the universally quantified variables have been skolemized - i.e. replaced by new constants - with a complete strategy of resolution.

Statment2 : if a formula F is an instance of an axiom of P it is valid (this is decidable).

Statment3 : equality axioms (1.5) can be used to deduce new formulas. For example : DEN(P) $\models$ p(t), t beeing any term and p a 1-any predicate, is equivalent to DEN(P) $\models$ x = t $\Rightarrow$ p(x) (the converse - i.e. DEN(P) $\models$ p(x) $\Rightarrow$ x = t - does not hold), because EQ $\models$ p(t) $\Leftrightarrow$ (x = t $\Rightarrow$ p(x)).

Statment4 : a formula F of the form $\overset{AND}{\underset{p}{}} \forall\ (p(\bar{x}) \rightarrow S^p[\bar{x}])$, where $\bar{x}$ denotes a sequence of distinct variables in place of the arguments of p, is valid if **S** is inductive in DEN(P).

Statment5 : if COMP(P) $\models$ F then DEN(P) $\models$ F which is obvious as DEN(P) is a model of COMP(P). This statment may be used when there is no inductive specification (see (7) below).

(3.16.1) Example : addition on the standard model of natural integers.

The program P is the following :

(c1) int(zero) ←
(c2) int(s(X)) ← int(X)
(c3) plus(zero, X, X) ←
(c4) plus(s(X), Y, s(Z)) ← plus(X, Y, Z).

One wants to prove that the following statment holds in DEN(P) (commutativity of plus)

(1) $int(X) \wedge int(Y) \wedge int(Z) \wedge plus(X, Y, Z) \Rightarrow plus(Y, X, Z)$

First of all notice that :

(2) $DEN(P) \models plus(X, Y, Z) \Rightarrow int(X)$

as the specification reduced to int (X) is inductive, i.e. :

$DEN(P) \models int(zero)$ and $DEN(P) \models int(X) \Rightarrow int(s(X))$ trivially.

Moreover :

(3) $DEN(P) \models plus(X, Y, Z) \wedge int(Y) \Rightarrow int(Z)$

as $S^{plus}(X, Y, Z) : int(Y) \Rightarrow int(Z)$ is inductive, i.e. :

$DEN(P) \models int(X) \Rightarrow int(X)$ trivially and

$DEN(P) \models (int(Y) \Rightarrow int(Z)) \Rightarrow (int(Y) \Rightarrow int(s(Z))$

as $int(Z) \Rightarrow int(s(Z))$ is a variant of an axiom.

Thus to prove the proposition (1), using (2) and (3), reduces to prove (4).

(4) $DEN(P) \models plus(X, Y, Z) \wedge int(Y) \rightarrow plus(Y, X, Z)$

Let us prove that $S^{plus}(X, Y, Z) : int(Y) \rightarrow plus(Y, X, Z)$ is inductive, i.e. that

(5) $DEN(P) \models int(X) \rightarrow plus(X, zero, X)$ and

$DEN(P) \models (int(Y) \rightarrow plus(Y, X, Z)) \rightarrow (int(Y) \rightarrow plus(Y, s(X), s(Z)))$

or better :

(6) $DEN(P) \models plus(X, Y, Z) \rightarrow plus(X, s(Y), s(Z))$ after a variables renaming.

The validity of (5) follows from the inductivity of $S^{int}(X) : plus(X, zero, X)$ i.e. :

$DEN(P) \models plus(zero, zero, zero)$ and

$DEN(P) \models plus(X, zero, X) \Rightarrow plus(s(X), zero, s(X))$

both are instances of clauses.

The validity of (6) follows from the inductivity of $S^{plus}(X, Y, Z) : plus(X, s(Y), s(Z))$ with the clauses of plus :

$DEN(P) \models plus(zero, s(X), s(X))$ and

$DEN(P) \models plus(X, s(Y), s(Z)) \Rightarrow plus(s(X), s(Y), s(s(Z))$

both are instances of clauses.

It is important to remark that in DEN(P) the formula plus(X, zero, X) is not valid (as it does not belong to DEN(P)), but only (5) is valid with any kind of term base. Note also the validity of :

$(plus(X, Y, Z) \wedge Y = zero) \Rightarrow X = Z$

as $S^{plus}(X, Y, Z) : Y = zero \Rightarrow X = Z$ is inductive.

Here is an other example :

(3.16.2) Example : list permutation

P is the following :

(c1) perm([ ], [ ]) ←

(c2) perm([A |L], [B|M]) ← perm(N, M), extract([A|L], B, N)

(c3) extract([A|L], A, L) ←

(c4) extract([A|L], B, [A|M]) ← extract(L, B, M)

(c5) list([ ])

(c6) list([A|L]) ← list(L)

Let us show that in DEN(P) all the arguments of "perm" are lists.

$$DEN(P) \models \forall [(perm(X, Y) \Rightarrow (list(X) \wedge list(Y))] \wedge$$
$$\forall [(extract(X, Y, Z) \wedge (list(Z) \Rightarrow list(X))]$$

as the specification :

$$\{ S^{perm} : list(perm1) \wedge list(perm2)$$
$$S^{extract} : list(extract3) \Rightarrow list(extract1) \}$$

is inductive (this has been proven in (3.8) with $\mathbb{L}$ as model, we prove it with DEN(P) as model using a rather automatic method) :

in c1 : DEN(P) |= list([ ]) ∧ list([ ])

in c2 : DEN(P) |= (list(N) ⇒ list([A|L])) ∧ list(N) ∧ list(M)
⇒ list([A|L])) ∧ list([B|M])

obvious

in c3 : DEN(P) |= list(L) ⇒ list([A|L])

in c4 : DEN(P) |= (list(M) ⇒ list(L)) ⇒ (list([A|M]) ⇒ list([A|L]))

This can be proven using the valid property :

(7) DEN(P) |= list([A|M]) ⇒ list(M)

but the property ∀ A, M (list1 = [A|M] ⇒ list(M)) is not inductive.

It is a consequence of the strongest specification S^{list} which is :

$$\exists\, n, A_1, \ldots A_n \quad list1 = [A_1, \ldots, A_n].$$

This example shows some limitations of the partially automatized method using statments 1 to 4 : such statments can be used to prove the validity of a formula as long as subformulas can be expressed in the form of an inductive specification.

The utility of such a proof method depends on the relationship between the domains of interest and DEN(P). It is necessary that the domains of interest are described by a term algebra isomorphic to the Herbrand base used to describe DEN(P). Then if the term interpretation of the predicates of P is exactly DEN(P), properties valid in DEN(P) correspond to properties valid in the considered interpretation. This is related to the idea of correctness and completeness of the program P w.r.t. some specification defined on some term interpretation.

In particular the proof of the formula (1) shows that the predicate "plus" is commutative in $\mathbb{N}$, the standard model of the natural integers (it is not true in $\mathbb{E}$ (1.4)). It has been assumed that all the

integers are represented in the denotation of the program. For example the same proof of valididty of the formula (1) :

$int(X) \wedge int(Y) \wedge int(Z) \wedge plus(X, Y, Z) \Rightarrow plus(X, Y, Z)$ could have been successfully performed on the following (obviously incomplete) program !

(c1) int(zero) ←
(c3) plus(zero, X, X) ←

(3.17) Conclusion on the inductive proof method

Let us summarize the results obtained in this section in three figures. We recall the definitions with the unique interpretation view (sections 3.1 to 3.14)

(val1) : $\mathbb{D}$ with $PTR_{\mathbb{D}}(P) \models \overset{AND}{\underset{p}{}} p(\bar{x}) \Rightarrow S^p[\bar{x}]$

(val2) : $DEN(P) \models \overset{AND}{\underset{p}{}} p(\bar{x}) \Rightarrow S^p[\bar{x}]$
(computational validity)

(proof1) : $\exists\, S'$ such that $\mathbb{D} \models P[S']$ and $\mathbb{D} \models S' \Rightarrow S$

Results are summarized on figure 3.17.1 (arrow denotes an implication) :

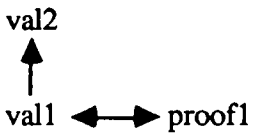


Figure 3.17.1 : results with any interpretation

If $\mathbb{D}$ is a term model both validity definitions become equivalent leading to Figure 3.17.2.

Figure 3.17.2 : results with term interpretations

The equivalence is conditioned by the existence of a formula specifying $PTR_{\mathbb{D}}(P)$.

With the axiomatic view (sections 3.15 and 3.16) :

(val1') : for all $\mathbb{D}$ s.t. $\mathbb{D} \models Ax$, $PTR_{\mathbb{D}}(P) \models \overset{AND}{\underset{p}{}} p(\bar{x}) \Rightarrow S^p[\bar{x}]$

(val2') : for all $p(\bar{t})$ in DEN(P), $Ax \models S^p[\bar{t}]$

(proof1') : $\exists\, S'$ such that $Ax \models P[S']$ and $Ax \models S' \Rightarrow S$

(proof2) : $Ax \cup COMP(P) \models \overset{AND}{\underset{p}{}} p(\bar{x}) \Rightarrow S^p[\bar{x}]$

results are summarized on Figure 3.17.3.

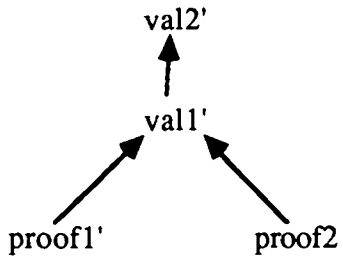


Figure 3.17.3 : results with the axiomatic view

Some more justifications are given in the annex.

4 - Proof method with annotations

The practical usability of the proof method of theorem (3.5) suffers from its theoretical simplicity : the inductive specifications S' to be found to prove the validity of some given specification S will need complex formulas S'^p since we associate only one for each p in PRED. It is also shown in [CD 88] that S' may be exponential in the size of the DCP (to show this result one can use the DCP's transformation into attribute grammars as in [DM 85]). The proof method with annotations is introduced in order to reduce the complexity of the proofs : the manipulated formulas are shorter, but the user has to provide the organization of the proof i.e. how the annotations are deducible from the others. These indications are local to the clauses and described by the so called logical dependency scheme. It remains to certify the consistency of the proof, i.e. that a conclusion is never used to prove itself. Fortunately this last property is decidable and can be verified automatically, using the Knuth algorithm [Knu 68] or its improvements [DJL 86].

(4.1) Definition : annotations of a DCP

Given a DCP P, an annotation is a mapping Δ assigning to every p in PRED a finite set of formulas or assertions $\Delta(p)$ built as in definition (2.7). It will be assumed that assertions are defined on $(L, \mathbb{D})$.

The set $\Delta(p)$ is partitioned into two subsets $I\Delta(p)$ (the set of the inherited assertions of p) and $S\Delta(p)$ (the set of the synthesized assertions of p).

The specification S_Δ associated with Δ is the family of formulas :

$$\{ S^p{}_\Delta : \text{AND } I\Delta(p) \Rightarrow \text{AND } S\Delta(p) \}_{p \in \text{PRED}}$$

(4.2) Definition : validity of an annotation Δ for a DCP P.

An annotation Δ is valid for a DCP P iff for all p in PRED in every proof-tree T of root $p(t_1, ..., t_{np})$: if $\mathbb{D} \models \text{AND } I\Delta(p)\ [t_1, ..., t_{np}]$ $(np = \rho(p))$ then every label $q(u_1, ..., u_{nq})$ $(nq = \rho(q))$ in the proof-tree T satisfies : $\mathbb{D} \models \text{AND } \Delta(q)\ [u_1, ..., u_{nq}]$.

In other words, an annotation is valid for P if in every proof-tree whose root satisfies the inherited assertions, all the assertions are valid at every node in the proof-tree, hence the synthesized assertions of the root.

(4.3) Proposition : validity of S_Δ

If an annotation Δ for the DCP P is valid for P, then S_Δ is valid for P.

Proof : It follows from the definition of S_Δ , the definitions of validity of an annotation (4.2) and of a specification (2.8).

Note that S_Δ can be valid but not inductive (see example (4.14), second part).

We shall give sufficient conditions insuring the validity of an annotation and reformulate the proof method with annotations. This formulation is slightly different from that given in [CD 88]. The introduction of the proof-tree grammar is a way of providing a syntaxic formulation of the organization of the proof.

(4.4) Definition : Proof-tree grammar (Gp)

Given a DCP P = < PRED, FUNC, CLAUS >, we denote Gp and call it the proof-tree grammar of P, the abstract context free grammar < PRED, RULE > such that RULE is in bijection with CLAUS and r of RULE has profile $< r_1 r_2 \ldots r_m, r_0 >$ iff the corresponding clause in CLAUS is $c : r_0(\ldots) \leftarrow r_1(\ldots), \ldots, r_m(\ldots)$.

Clearly a (syntax) tree in Gp can be associated to every proof-tree of P. But not every tree in Gp is associated a proof-tree of P.

(4.5) Definition : Logical Dependency Scheme for Δ (LDS_Δ)

Given a DCP P and an annotation Δ for P, a logical dependency scheme for Δ is $LDS_\Delta = < Gp, \Delta, D >$ where Gp = < PRED, RULE > is the proof-tree grammar of P and D a family of binary relations defined as follows.

We denote for every rule r in RULE of profile $< r_1 r_2 \ldots r_m, r_0 >$
$W_{hyp}(r)$ (resp. $W_{conc}(r)$) the sets of the hypothetic (resp. conclusive) assertions which are :
$W_{hyp}(r) = \{\varphi_k \mid k = 0, \varphi \in I\Delta(r_0) \text{ or } k > 0, \varphi \in S\Delta(r_k)\}$
$W_{conc}(r) = \{\varphi_k \mid k = 0, \varphi \in S\Delta(r_0) \text{ or } k > 0, \varphi \in I\Delta(r_k)\}$

where φ_k is φ in which the free variables free $(\varphi) = \{p_1, \ldots, p_n\}$ have been renamed into free $(\varphi_k) = \{p_{k1}, \ldots, p_{kn}\}$.

The renaming of the free variables is necessary to take into account the different instances of the same predicate (if $r_i = r_j = pr$ in a clause for some different i and j) and thus different instances of the same formula associated with pr.

$D = \{D(r)\}_{r \in RULE}$, $D(r) \subseteq W_{hyp}(r) \times W_{conc}(r)$.

From now on we will use the same name for the relations D(r) and their graph. For a complete formal treatment of the distinction see for example [CD 88]. We denote by hyp (φ) the set of all formulas ψ such that $(\psi, \varphi) \in D(r)$ and by assoc $(\varphi) = p(t_1, \ldots, t_n)$ the atom to which the formula is associated by Δ in the clause c corresponding to r.

(4.6) Example : annotation for example (2.2) and specification S_2 (2.9)

$\Delta(plus) = I\Delta(plus) \cup S\Delta(plus)$

$I\Delta(plus) = \{\varphi : ground(Plus3)\}$

$S\Delta(plus) = \{\psi : ground(Plus1), \delta : ground(Plus2)\}$

$S_\Delta^{plus} = S_2^{plus}$

G_{plus} : PRED = {plus}

RULE = $\{r_1 \in < \varepsilon, plus>, r_2 \in < plus, plus >\}$

D : $D(r_1) = \{\varphi_0 \rightarrow \delta_0\}$

$D(r_2) = \{\varphi_0 \rightarrow \varphi_1, \psi_1 \rightarrow \psi_0, \delta_1 \rightarrow \delta_0\}$ (see the scheme below)

$W_{\underline{hyp}}(r_1) = \{\varphi_0\}$, $W_{\underline{conc}}(r_1) = \{\psi_0, \delta_0\}$

$W_{\underline{hyp}}(r_2) = \{\varphi_0, \psi_1, \delta_1\}$, $W_{\underline{conc}}(r_2) = \{\varphi_1, \psi_0, \delta_0\}$

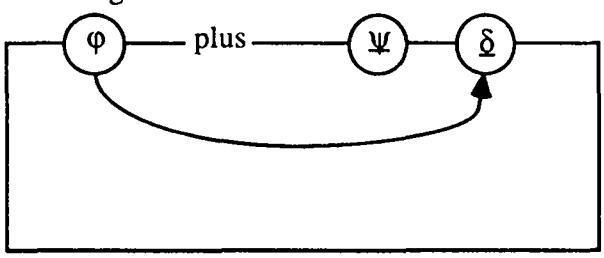
Note that in r_2 for example :

φ_0 = ground(Plus03)

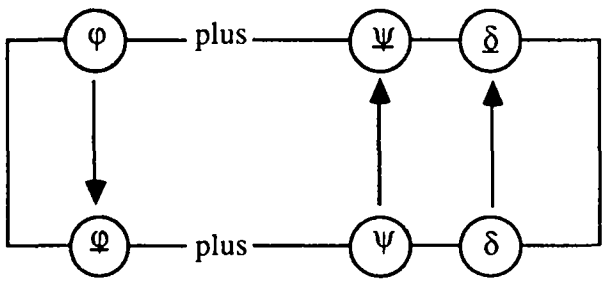
φ_1 = ground(Plus13)...

In order to simplify the presentation of D we will use schemes as in [CD 88] representing the rules in RULE and the LDS of Δ. Elements of $W_{\underline{conc}}$, will be underlined. Inherited (synthesized) assertions are written on the left (right) hand side of the predicate name. Indices are implicit : 0 for the root, 1... to n following the left to right order for the sons.

r 1 :



r 2 :



(4.7) Definition : Purely-synthesized LDS, well-formed LDS.

A LDS for Δ is <u>purely-synthesized</u> iff $I\Delta = \emptyset$, i.e. there are no inherited assertions.

A LDS for Δ for P is <u>well-formed</u> iff in every tree t of Gp the relation of the induced dependencies D(t) is a partial order (i.e. there is no cycle in its graph).

To understand the idea of well-formedness of the LDS it is sufficient to understand that the relations D(r) describe dependencies between instances of formulas inside the rules r. Every tree t of G_P is built with instances of rules r in RULE, in which the local dependency relation D(r) defines dependencies between instances of the formulas attached to the instances of the non-terminals in the rule r. Thus the dependencies in the whole tree t define a new dependency relation D(t) between instances of formulas in the tree. A complete treatment of this question can be found in [CD 88]. We recall here only some important results [see Knu 68, DJL 88 for a survey on this question] :

(4.8) Proposition : known properties of the LDS's

- The well-formedness property of an LDS in decidable.
- The well-formedness test is intrinsically exponential.
- Some non trivial subclasses of LDS can be decided in polynomial time.
- A purely-synthesized LDS is (trivially) well-formed.

(4.9) Definition : Soundness of a LDS for Δ.

A LDS for $\Delta < G_p, \Delta, D >$ is sound iff for every r in RULE and every
φ in $W_{conc}(c)$ with $\underline{assoc}(\varphi) = q(u_1, ..., u_{nq})$ the following holds :
$\mathbb{D} \models \mathrm{AND}\{ \psi[t_1, ..., t_{np}] \mid \psi \in \underline{hyp}(\varphi)$ and $\underline{assoc}(\psi) = p(t_1, ..., t_{np}) \} \Rightarrow \varphi[u_1, ..., u_{nq}]$
(Note that the variable q_i (p_i) in a formula φ (ψ) is replaced by the corresponding term u_i (t_i)).

(4.10) Example : The LDS given in example (4.6) is sound.

In fact it is easy to verify that the following holds in $\mathbb{T}$:

in r1 :	ground(X)	⇒	ground(X)
	ground(zero)		
in r2:	ground(sX)	⇒	ground(X)
	ground(Y)	⇒	ground(Y)
	ground(Z)	⇒	ground(sZ)

(4.11) Theorem : (Validity of an annotation)

Given an annotation Δ for a DCP P, Δ is valid for P if there exists a sound and well-formed LDS_Δ for Δ for P.

Proof : we follow the sketch given in [Der 83]. Given a proof-tree t, the relation D(t) is a partial order on the instances of the assertions associated to the nodes of t. The proof is done by induction on this order. The minimal elements of D(t) are two folds : inherited formulas of the root (in IΔ (root)) or formulas φ associated to a node n with no antecedent following the local dependency scheme corresponding to the contexte rule c_1 if φ is inherited or the root subtree c_2 if φ is synthesized (see Figure 4.11.1). By hypothesis (sound Δ), these latter formulas φ belong to $W_{conc}(c)$ — c beeing the rule — with assoc $(\varphi) = q(u_1, ..., u_n)$ and $\mathbb{D} \models \varphi[u_1, ..., u_n]$.

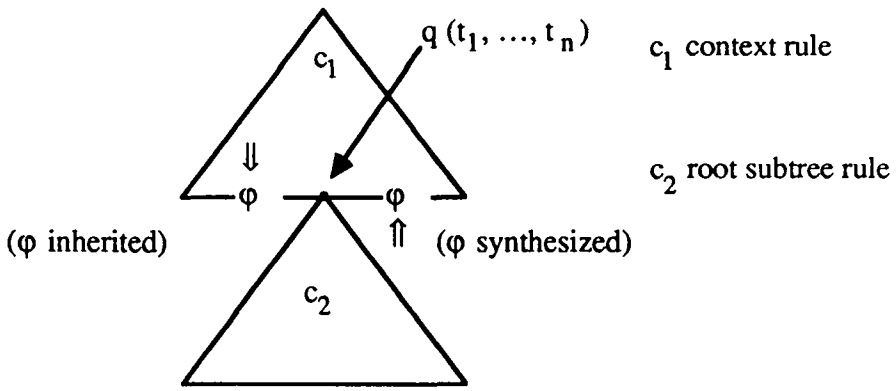


Figure 4.11.1 : minimal elements in D(t)

By definition of a proof-tree, q (t_1, ..., t_n) is an instance of q (u_1, ..., u_n) in c1 and c2, hence $\mathbb{D} \models q\ (t_1, \ldots, t_n)$.

By the definition of validity one assumes who that the instances of the formulas at the root of t are also valid by $\mathbb{D}$. Notice that there are two notions of "instance" : instance of the formulas at some node (here the root) — i.e. free variables are renamed —) (here the root) and instance of the formulas by a substitution — i.e. renamed free variables are replaced by terms — which is valid, as every instance (by a substitution) of a valid formula is valid in $\mathbb{D}$.

Now consider any instance of a formula φ in D (t) which is not minimal. Taking the formulas which depends on φ, they correspond to an instance of a local dependency scheme which is from the context rule c_1 if φ is inherited or in the root subtree rule c_2 if φ is synthesized (see Figure 4.11.1). By the soundness hypothesis of LDS_Δ the formula φ [t_1, ..., t_n] is valid in $\mathbb{D}$ because a proof-tree is built with clauses instances and all the formulas φ depends on belong to the same rule. QED.

(4.12) Theorem (soundness and completeness of the annotation method for proving the validity of specifications) : we use the notations of (3.3) and (3.5).

A specification **S** on (L, $\mathbb{D}$) is valid iff it is weaker than the specification $\mathbf{S}_\Delta$ of an annotation Δ with a sound and well-formed LDS.

i.e. :

1) there exists a sound and well-formed LDS_Δ.

2) $\mathbb{D} \models \mathbf{S}_\Delta \Rightarrow \mathbf{S}$.

Proof (soundness) follows from theorem (4.11).
(completeness) follows from the fact that $\mathbf{S}_{P,\mathbb{D}}$ is a purely synthesized (thus well-formed) sound annotation.

We complete this presentation by giving some examples.

(4.13) Example (4.10) continued.

The LDS is sound and well-formed, thus $\mathbf{S}_\Delta^{plus} = \mathbf{S}_2^{plus}$ is a valid specification.

(4.14) Example : A valid specification which is not inductive

We borrow from [KH 81] an example given here in a logic programming style : it computes multiples of 4.

c1 : fourmultiple (K) $\leftarrow$ p(zero, H, H, K).

c2 : p(F, F, H, H) $\leftarrow$

c3 : p(F, sG, H, sK) $\leftarrow$ p(sF, G, sH, K)

$S^{fourmultiple}$: $\exists$ N, N $\geq$ 0 $\wedge$ Fourmultiple1 = 4$*$N

L, $\mathbb{D} = \mathbb{D}_1$ as in (2.9) enriched with $*$, ≥ 0 etc...

The following annotation Δ is considered in [KH 81] :

$I\Delta(\text{fourmultiple}) = \emptyset$, $S\Delta(\text{fourmultiple}) = \{ S^{\text{fourmultiple}} \} = \{\delta\}$

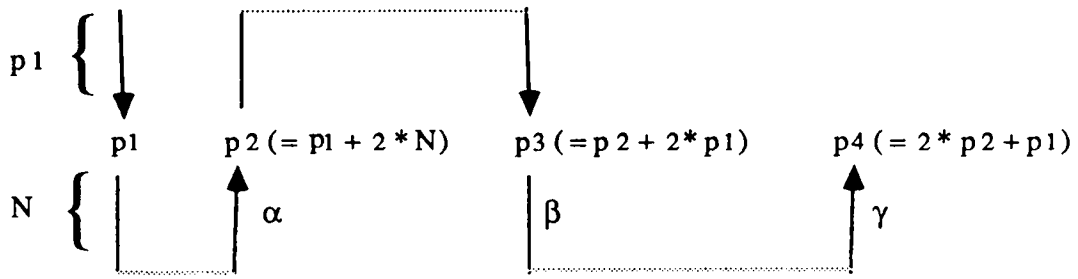
$I\Delta(p) = \{\beta\}$ $S\Delta(p) = \{\alpha, \gamma\}$

α : $\exists N, N \geq 0 \wedge p2 = p1 + 2 * N$

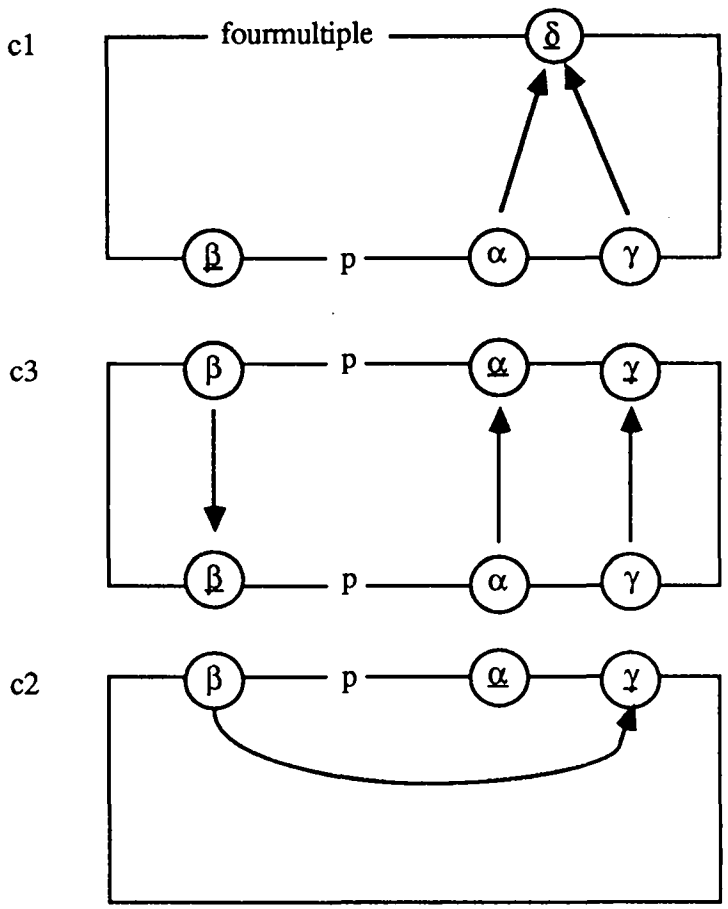
β : $p3 = p2 + 2 * p1$

γ : $p4 = 2 * p2 + p1$

The assertions can be easily understood if we observe that such a program describes the construction of a "path" of length 4$*$N and that p1, p2, p3 and p4 are lengths at different steps of the path as shown in the following figure :



The LDS for Δ is the following :



The LDS is sound and well-formed. For example it is easy to verify that the following fact holds in $\mathbb{D}_1$:

in c1 :

$(\alpha_1 \wedge \gamma_1 \Rightarrow \mathbf{S}_0^{\text{fourmultiple}})$ that is : $\exists$ N, N $\geq 0 \wedge$ H = zero + 2∗N $\wedge$ K = 2∗H+ zero

$\Rightarrow \exists$ N, N $\geq 0 \wedge$ K = 4∗N

(β_1) that is : H = H + 2∗zero

in c2 :

$(\beta_0 \Rightarrow \gamma_0)$ that is : H = F + 2∗F $\Rightarrow$ H = 2∗F + F

(α_0) that is : $\exists$ N, N $\geq 0 \wedge$ F = F + 2∗N (with N = 0)

etc...

Note that as $\mathbf{S}_\Delta$ is inductive, this kind of proof modularization can be viewed as a way to simplify the presentation of the proof of $\mathbf{S}_\Delta$.

Now we consider on the same program a non inductive valid specification ξ defined on L_2, $\mathbb{D}_2$ (2.9) :

$\xi^{\text{fourmultiple}}$: ground (fourmultiple1)

ξ^{p} : [ground(p1) $\wedge$ ground(p3)] $\Rightarrow$ [ground(p2) $\wedge$ ground(p4)]

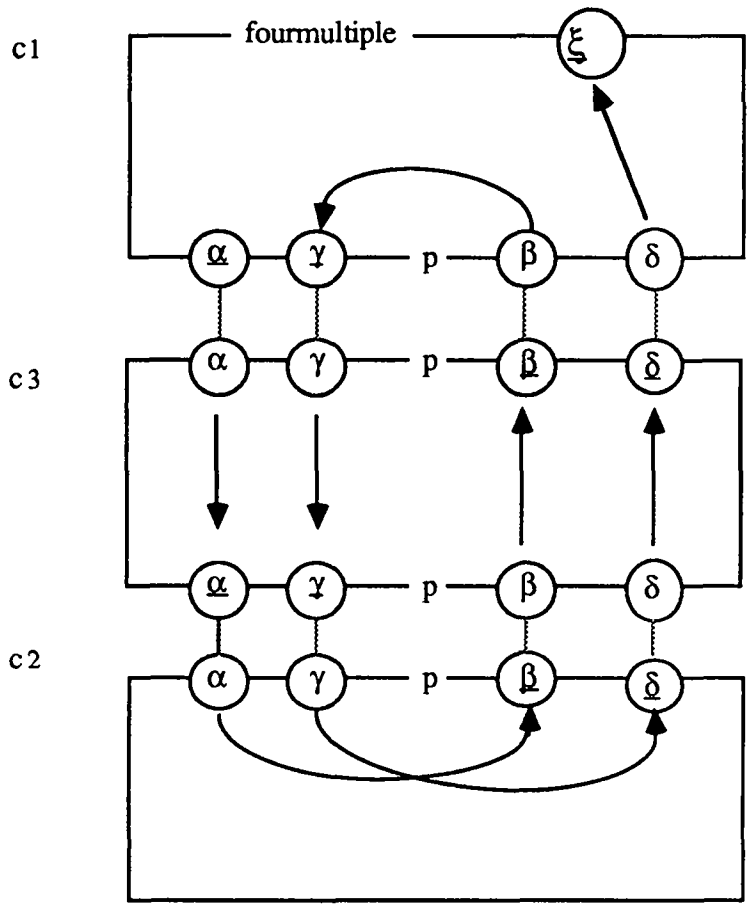
The specification clearly is valid but not inductive since the following does not hold with $\mathbb{D}_2$ (term-algebra) in c1

$$\mathbb{D}_2 \not\models \xi_1^p \Rightarrow \xi_0^{\text{fourmultiple}}$$

i.e.

$$\mathbb{D}_2 \not\models [\ (\text{ground(zero)} \wedge \text{ground(H)}) \Rightarrow (\text{ground(H)} \wedge \text{ground(K)})\ ] \Rightarrow \text{ground(K)}$$

But it is easy to show that the following LDS is sound and well-formed :



$I\Delta(\text{fourmultiple}) = \emptyset$, $S\Delta(\text{fourmultiple}) = \{\ \xi^{\text{fourmultiple}}\}$

$I\Delta(p) = \{\alpha, \gamma\}$, $S\Delta(p) = \{\beta, \delta\}$

α : ground(p1)

β : ground(p2)

γ : ground(p3)

δ : ground(p4)

Dotted lines help to observe that the LDS is well-formed (without circularities). The proofs are trivial.

Note that the corresponding inductive specification is $(\alpha \Rightarrow \beta) \wedge (\gamma \Rightarrow \delta)$. It is shown in [CD 88] how the inductive specification can be inferred from LDS_Δ.

Notice that this kind of proof of correctness corresponds to some kind of mode verification. It can be automatized for a class of programs identified in [DM 84] and experimentally studied in [Dra 87] (the class of simple logic programs). As shown in [DM 84] this leads to an algorithm of automatic

(ground) modes computation for simple logic programs which can compute (ground) modes which are not inductive.

(4.15) Power of the method of annotations

It has been shown in [CD 88] that the annotation method does not permit to prove more valid specifications than the fixpoint induction method does, hence it has the same theoretical power. This is achieved by showing that to a valid annotation with inherited and synthesized assertions it is possible to associate a purely synthesized one (hence inductive) built with the annotation. Furthermore it is shown that this inductive specification may have an exponential size (with regards to the size of the annotation). This shows that on one side the practical complexity of the proof may be improved by using annotations, and on the other side, from a practical point of view, it could be much more convenient to use many directional assertions rather than one inductive only.

In fact, even for relatively simple specification like: $S1 \Rightarrow S2$, it is quite natural to consider one inherited (S1) and one synthesized (S2). For example, assuming some recursive axiom at nodes 0 and 1, instate of proving a formula like : $(S1(1) \Rightarrow S2(1)) \Rightarrow (S1(0) \Rightarrow S2(0))$ which may be trublesome to handle, one just has to prove two formulas like: $S1(0) \Rightarrow S1(1)$ and $S2(1) \Rightarrow S2(0)$ for example. Furthermore the use of inductive assertion becomes totally unnatural if properties which depends on particular upper context inside the tree are considered. The example of section 5 uses such kind of property.

(4.16) An axiomatic view of the annotation method

All the results of this section still hold if one replaces the interpretation $\mathbb{D}$ by a set of axioms Ax, except theorem 4.12, that we can restate as follows :

(4.16.1) **Theorem** (Soundness of the annotation method-axiomatic view)

A specification **S** on L with axioms Ax is valid for P if it is weaker than the specification $\mathbf{S}_\Delta$ of an annotation Δ in L with a sound and well-formed LDS, i.e. :

1) There exists LDS_Δ well-formed such that :
   for every r in RULE and every in φ in $W_{\underline{conc}}(r)$ with $assoc(\varphi) = q\ (u_1, \ldots, u_{nq})$:
   $Ax \models AND\ \{\Psi(t_1, \ldots, t_n) \mid \Psi \in hyp\ (\varphi)$ and $assoc\ (\Psi) = p(t_1, \ldots, t_{nq})\} \Rightarrow \varphi\ [u_1, \ldots, u_{nq}]$

2) $Ax \models \mathbf{S}_\Delta \Rightarrow \mathbf{S}$.

Indeed the proof of theorem (4.11) with Ax in state of $\mathbb{D}$ is exactly the same. But as the completeness of the method (theorem 3.15.3) does not hold any more with the axiomatic view for purely synthesized LDS, the annotation method is not complete either, even if Ax is a complete axiom system.

5. Illustration of the proof method by annotations

One may think that the proof method by annotation is of limited interest since it has the same power as the fixpoint method in the case of root specifications and the exponential complexity of the inductive assertions rarely happens. In fact it is difficult to be convinced of the good value of the method if we only base ourselve on small examples. In turn if we consider larger and more realistic examples its superiority appears very naturally. We illustrate this claim by giving an example of a piece of a compiler for a toy language.

This example illustrates also the versatility of logic programming by the way the symbol table is handled in order to mantain partially known references. The example is developed step by step after we have given some unformal description of the used data structures (specifications and their interpretation). First we introduce a specification of the compiler by the definition of the source and object languages, and the mapping defining the translation. Second we give the data structures and the basic elements needed to understand the annotations which will be used. Then we develop step by step the program by adding new arguments and new annotations in the following order : analysed sentence, symbol table, addresses and generated code. Using the annotation proof method, the compiler will be formally (even if some part of the interpretation is not completely described) proven partially correct. The complete program (see P6) corresponds to the logic program resulting by the transformation described in 5.1 from a Definite Clause Grammar. Hence the program is structured by the initial context free grammar and additionnally there are auxiliary predicates used to describe manipulations of the symbol table. The proof of partial correctness of this additional program will not be given here as its specification is inductive and it is not difficult to do it. However the whole proof assumes that the free variables have a defined type. Moreover for some of them some properties are assumed. For example it will be assumed that the items of a symbol table are without repetition. Such properties have to be proven on the global program. This will be done in a separate step (5.7).

The language is defined by the following context free grammar :
G :

prog	→	**li**	**li** stands for "list of instructions"
li	→	**li ; li**	
li	→	**lins**	**lins** stands for "(labeled) instruction"
lins	→	**lab : ins**	**lab** stands for "label"
lins	→	**ins**	
ins	→	**ins**	
ins	→	**if expr thengoto lab**	conditional jump

Underlined symbols are terminals ; **lab**, **ins** and **expr** are supposed generics; they are not described expicitely is order to keep this example simple. The language generated by G is denoted L(G). Notice that the grammar is ambiguous.
Here is a possible program :

Prog0 : a: **ins** ; **if expr thengoto** d ; **if expr thengoto** a ; d: **ins**

The purpose of the compiler is to produce the object code consisting of a list of pairs or triples of the form :

[number, **ins**] or [number, **expr**, number]

in the same order imposed by the sequentializing operator ";" in the source language and such that "number" is an instruction address in the generated code and that all the addresses 1, 2, 3, ... corresponding to the first "number" follow a consecutive order starting by 1).

We will not give a more formal definition as it seems clear enough that to any source program generated by **G**, say **P**, it corresponds a unique list of object instructions denoted **trans(P)**, assuming that all the references are solved. For example :
trans(Prog0) = [[1, ins], [2, expr, 4], [3, expr, 1], [4, ins]]

The annotations we will use in the proofs are defined on the interpretation $\mathbb{L}$ (section 1.4) enriched with some sorts and functions. We give here some of them :
trans ∈ <lists, lists> has domain L(G) (lists representing the source programs). Its result is a list which is the generated code.
l-table ∈ <lists, lists> its results is a list representing a label-table. A source program contains labels which can be stored in the label-table without repetition associated with the address of the instruction where they are defined. It will be assumed that a label is uniquely defined. For example to the source program :

(a: **ins** ; **if expr thengoto** d) ; **if expr thengoto** a

it corresponds the label-table : [[a, 1], [d,X]]

as this program contains two labels in which only one is defined, but the program :

a : **ins** ; a : **ins** will not have any translation as it is assumed in **trans** that different instructions have different address (error handling is not cnsidered here).

The "updating" of the label-tables will be defined by a logic program for which we just give the corresponding inductive assertion assuming it is well-typed. These assertions can be understood easely if one assumes that a label-table is a list of items of the form [label, integer] in which the integer represents the address of the instruction in the source program in which it is defined (it may be a variable also).

The label-table will be updated by the following relation :
test-or-incl
which adds the item to the table if it is not already in the table. It is defined here :

test-or-incl(T, I, T) ← is-already-in(T, I)
test-or-incl(T, I, [I | T]) ← is-not-already-in(T, I)

is-already-in([I | T], I) ←
is-already-in([I1 | T], I2) ← is-already-in(T, I2)

is-not-already-in([], I) ←
is-not-already-in([I1 | T], I2) ← diffI(I1, I2), is-not-already-in(T, I2)

diffI([E1, A1], [E2, A2]) ← diff(E1, E2)

It is not difficult to prove its correctness w.r.t. the following specification (which is inductive but not the strongest one) :

$\{ S^{\text{test-or-incl}}, S^{\text{is-already-in}}, S^{\text{is-not-already-in}}, S^{\text{diffI}}, S^{\text{diff}}, S^{\text{is}} \}$

S^{is} (is1, is2) : is1 $=_N$ is2 (This specification will be used later).

S^{diff} (diff1, diff2) : diff1 $\neq_T$ diff2

S^{diffI} (diffI1, diffI2) : car (diffI1) $\neq_T$ car (diffI2)

$S^{\text{is-already-in}}$ (iai1, iai2) : $\exists$ I, I $\in$ iai1 $\wedge$ I $=_{\text{item}}$ iai2
(iai1 interpreted as a list of items of a label table)

$S^{\text{is-not-already-in}}$ (inai1, inai2) : not ($\exists$ I, I $\in$ inai1 $\wedge$ car (I) $=_T$ car (inai2))

$S^{\text{test-or-incl}}$ (toi1, toi2, toi3) : $\forall$ I, I $\in$ toi3 $\Leftrightarrow$ (I $\in$ toi1 $\vee$ I $=_{\text{item}}$ toi2))
$\wedge$ noduplic (toi1) $\Rightarrow$ noduplic (toi3)

Some remarks have to be made on these specifications. First of all "diff" is not specified. One may assume here, even if it is not realistic, that it is given by a set of facts diff (a, b) in which a and b are any possible different labels appearing in a source program. "is" will not be defined either. It is assumed to be an addition table of the form : a is b + c where a, b and c are natural integers. Second the specifications are expressed in a many sorted language. As the program contain not explicitely typed variables, it is necessary to complete the proof of correctness by showing that the program is well typed. For example one will assume that test-or-incl satisfies also the specification : "if toi1 is a table and toi2 a label then toi3 is a table". This will guarantee in particular that when the specification diff1 $\neq_T$ diff2 is assumed, diff1 and diff2 are labels, or that I $=_{\text{item}}$ iai2 in $S^{\text{is-already-in}}$ holds on items, i.e. pairs of "label" and "number" which are tested respectively equal. This verification of well-typedness is given in the section 5.7 for the whole program P6 (section 5.6) with an annotation.

Now we start the proof with an annotation concerning the syntax directed compiler.

(5.1) P1 : syntactic analysis

The syntactic aspects of the translation consist of applying a wellknown transformation to the grammar G to get a non deterministic descendent analyser written with definite clauses [Col 79]. We recall here the transformation, but based on difference-lists, and prove its partial correctness using the interpretation $\mathbb{L}$ (1.4) enriched with some more functions.

Transformation

If the rule has the form :

$x \rightarrow t_1, t_2, ..., t_n$, $n \geq 0$, t_i terminal symbol, then generate the clause :

(1) $x([t_1, ..., t_n \mid L] - L) \leftarrow$

One assumes that the source program is represented by a difference-list in which each element represents a token of the program (i.e. a terminal in the grammar of the source program).
If the rule has the form:

$x_0 \rightarrow w_0\ x_1\ w_1 \ldots x_k\ w_k \ldots x_n\ w_n$, then generate the clause :

(2) $x_0([w_0 \mid L_0] - L_n) \leftarrow x_1(L_0 - [w_1 \mid L_1]), ..., x_k(L_{k-1} - [w_k \mid L_k]), ..., x_n([L_{n-1} - [w_n \mid L_n])$

where $[w_k \mid L_k]$, $0 \leq k \leq n$ stands for $[t_1, ..., t_n \mid L_k]$ if $w_k = t_1 \ldots t_n$ and L_k only if w_k is empty.

Partial Correctness :

To every non-terminal symbol **x** of **G** we associate the following specification that we state informally :

S_1^x = **X1** is a difference-list which represents a terminal string derived from **x**

This can be stated formally using the usual transitive closure of the string derivation relation:

$S_1^x = x \xrightarrow{+}$ **repr(X1)** (**X1** denotes the first argument of **x**)

It is easy to show that $S_1 = \{ S_1^x \mid$ for all non-terminals **x** of **G**$\}$ is inductive on $\mathbb{L}$, hence valid.

- It is obvious for **(1)**.
- Fig. 5.1 illustrates the proof for **(2)** :

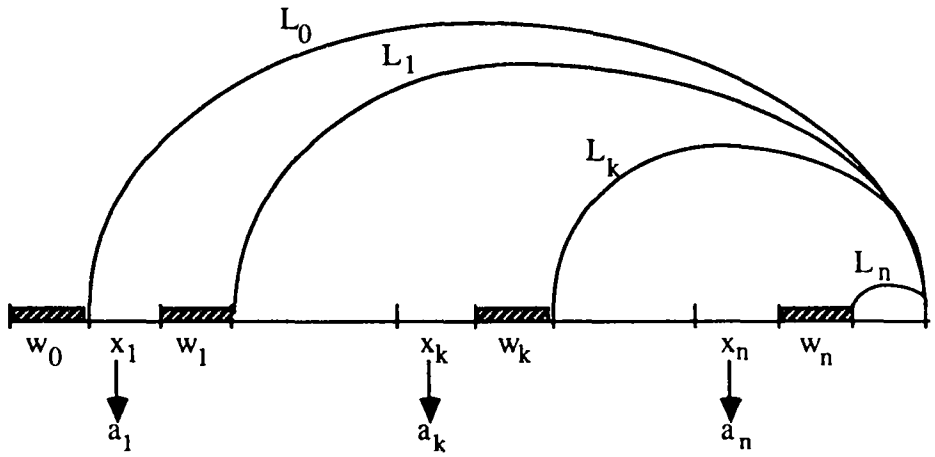


Figure 5.1 : proof of transformation rule 2

The extra variables $L_0, L_1, ..., L_n$ are used to define the unknown strings $a_1, ..., a_n$ and if, by hypothesis $x_i \xrightarrow{+} a_i$ then the conclusion $x_0 \xrightarrow{+} w_0\ a_1\ w_1 \ldots a_k\ w_k \ldots a_n\ w_n$ holds.
The completeness is not proven here. It results in particular from the fact that there are as many clauses than there are rules in the grammar **G**.
Applying this transformation to the grammar **G** leads to the following program (after simplifications which preserve obviously the partial correctness and the completeness) :

P1

prog(L) ← li(L)

li(L0- L2) ← li(L0- L1), li(L1- L2)
li(L) ← lins(L)

lins([**lab**, _ | L0]- L1) ← ins(L0- L1)
lins(L) ← ins(L)

ins([**ins** | L]- L) ←
ins([**if**, **expr**, **thengoto**, **lab** | L]- L) ←

From now on we will introduce new arguments one after the other such that the annotations may be introduced progressively. The first refinement consists in a new argument which will be removed at the end. It is needed to make a completely formal proof.

(5.2) P2 : adding the beginning of a sentence :

In order to deal with labels and to build a label-table (to store and update backward and forward references) we want to be able to state something like : "all the defined labels of some part of the source program are in the table". Thus we introduce an argument whose purpose is to capture what has been already analysed "before" a non-terminal x following the left to right order of the terminal sentences derived from the grammar axiom "prog". It is the purpose of the new first argument added to all the predicates.

P2

prog(M-L) ← li(M- M, M- L)

li(M- L0, L0- L2) ← li(M- L0, L0- L1), li(M- L1, L1- L2)
li(M, L) ← lins(M, L)

lins(M- N, [lab, : | L0]- L1) ← ins(M- L0, L0- L1)
lins(M, L) ← ins(M, L)

ins(M, [ins | L]- L) ←
ins(M, [if, expr, thengoto, lab | L]- L) ←

In state of doing the proof for this particular program, we will define this new argument on the general transformed programs. Thus we state again the transformation, but with a new argument which represents the sentence derived from the grammatical axiom "before" the current non-terminal. We first add a non-terminal "axiom" to G and a unique rule 0 as follows :

(0) **axiom(M - L)** ← **x(M - M, M - L)** (for some x)

(1) **x(M, [t_1, ..., t_n | L] - L])** ←

(2) **x_0(M -N, [w_0 | L_0] - L_n)** ← **x_1(M- L_0, L_0 - [w_1 |L_1]), ...,**
x_k(M- L_{k-1}, L_{k-1}- [w_k | L_k]), ...,
x_n(M- L_{n-1}, L_{n-1}- [w_n | L_n])

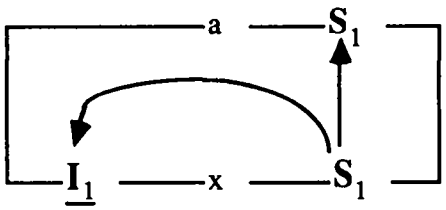
The first argument plays the role of the "beginning" of the sentences derived from "axiom". Thus we want to prove something like "in every proof-tree of root "axiom", at every node x different from "axiom", axiom $\overset{+}{\rightarrow}$ repr(**X1**) repr(**X2**) α holds" (**X1** is the first argument of x and **X2** the second — remember that **x1** is the terminal string derived from x —, α some tail of the derived string).

By the previous results concerning the inductive assertion S_1^x = x $\overset{+}{\rightarrow}$ **repr(X2)** holds. Thus it is sufficient to consider an inherited assertion associated to the non-terminal symbols different from "axiom":

I_1^x : $\exists\, \alpha$ axiom $\xrightarrow{+}$ repr(**X1**) x α and $\exists$ M, N, L **X1** = M-N and **X2** = N-L

We prove it by following the logical dependency scheme given in Fig. 5.2. (Note that α may be any sequence of terminals and non terminals).

Axiom rule :



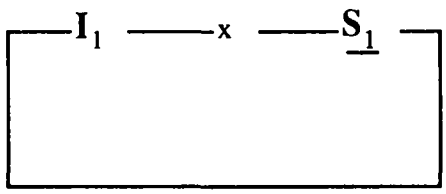
I_1^{x1} follows from S_1^{x1} :

$$x \xrightarrow{+} \text{repr(M-L)} \Rightarrow \text{axiom} \xrightarrow{+} \text{repr(M-L)} \Rightarrow \text{axiom} \xrightarrow{+} \text{repr(M-M) repr(M-L) } \alpha$$

S_1^{x1} $\qquad$ S_1^{axiom} $\qquad$ repr(M-M) is the empty sentence

(α is empty).
Second part of the assertion is obvious.

Terminal rule :



Nothing to prove as $I_1(0)$ is inherited.

General rule (2) :

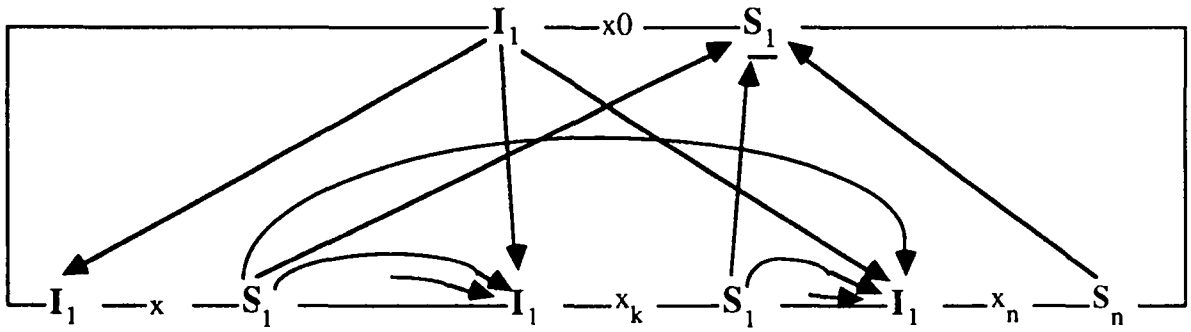
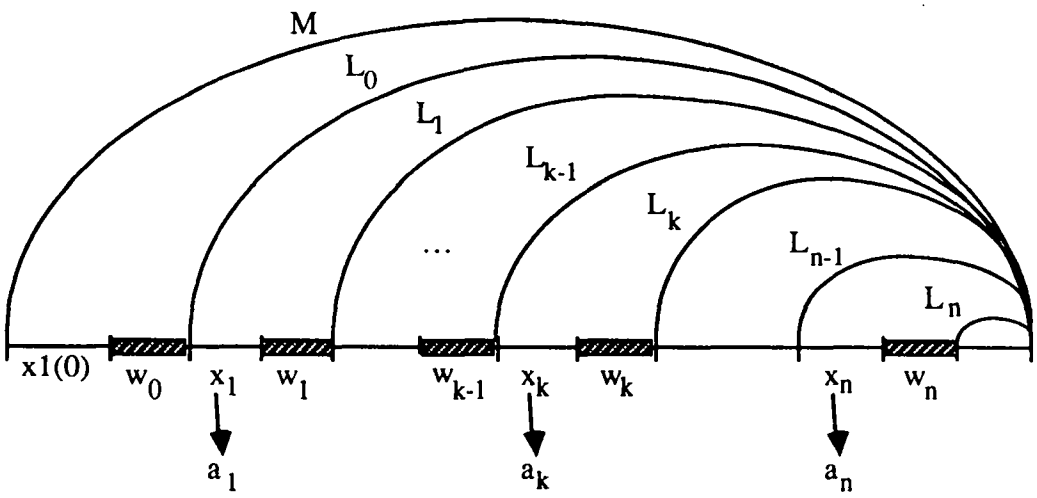


Figure 5.2 : proof schemes for I_1

For the proof we extend the scheme given for the proof of S_1 in (transformation 2) with M and **x1(0)** the first argument of the root — it has the form M - L by hypothesis. By hypothesis also :

$$\text{axiom} \xrightarrow{+} \text{repr(M- [w}_0 \text{ | L}_0\text{]) repr([w}_0 \text{ | L}_0\text{] - L}_n \text{)}$$



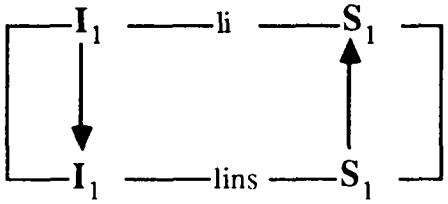
By hypothesis x1(0), i.e. M- $[w_0 \mid L_0]$ by the second part of the assertion, corresponds to a derivation of the form $\text{axiom} \xrightarrow{+} \mathbf{x1(0)}\ x\ \alpha$. As

$$x_0 \xrightarrow{+} w_0\ x_1\ w_1 \ldots x_k\ w_k \ldots x_n\ w_n \quad \text{by hypothesis,}$$

$\text{axiom} \xrightarrow{+} \mathbf{x1(0)}\ w_0\ a_1\ w_1 \ldots a_k\ w_k \ldots a_n\ w_n\ \alpha$ and the $\mathbf{I_1}^{x_k}$'s, $1 \le k \le n$, hold as M-L_{k-1} corresponds to the derived sentence from "axiom", preceeding x_k. Note that the forms M-N for **X1** and N-L for **X2** are preserved.

Notice that in fact only assertions $\mathbf{S_1(j)}$, $j < k$ are needed to prove $\mathbf{I_1(k)}$.

The program P2 has been simplified as P1 (variables replace identical difference-lists) in a trivial way. The adaptation of the logical dependency schemes to the case of P2 is straightforward. For example, in the third rule, we get :



(5.3) P3 : adding the length of the source program

As we know there will be as many instructions in the generated code than there are instructions in the source program. Thus the address of some instruction can be defined by the length of the derived sentence "before" the instruction (+1), i.e. the address of the first instruction corresponding to the code generated for a non-terminal x is length(**X1**) + 1. One could use directly this information inside the program. But we want to avoid the explicit use of the first argument which will be removed later on, and moreover one wants to use simple operations only (addition in place of length). Hence we add two arguments to all non terminals in { li, lins, ins} denoted **A1** and **A2** (in state of **X3** and **X4** for mnemonic reasons) whose specifications are :

$\mathbf{I_2^x(X1, A1)}$: **A1** = length(repr(**X1**))+ 1

$\mathbf{S_2^x(X1, X2, A2)}$: **A2** = length(repr(**X1**) repr(**X2**))+ 1 for all x in { li, lins, ins}

P3

prog(M-L) ← li(M- M, M- L, 1, A2)

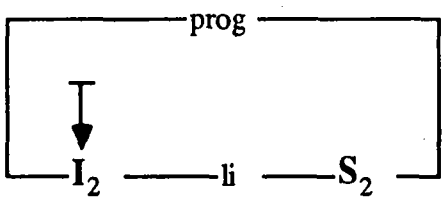
li(M- L0, L0- L2, A1, A2) ← li(M- L0, L0- L1, A1, A3), li(M- L1, L1- L2, A3, A2)
li(M, L, A1, A2) ← lins(M, L, A1, A2)

lins(M- N, [**lab**, : | L0]- L1, A1, A2) ← ins(M- L0, L0- L1, A1, A2)
lins(M, L, A1, A2) ← ins(M, L, A1, A2)

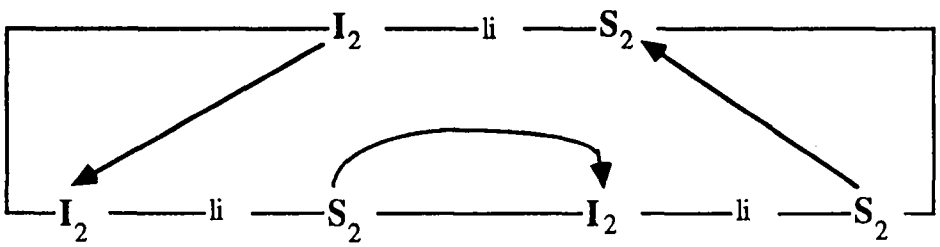
ins(M, [**ins** | L]- L, A1, A2) ← A2 is A1 + 1
ins(M, [**if**, **expr**, **thengoto**, **lab** | L]- L, A1, A2) ← A2 is A1 + 1

The annotation is proven sound by the dependency schemes of Fig.3. Most of them are obvious, following the definitions of the arguments.

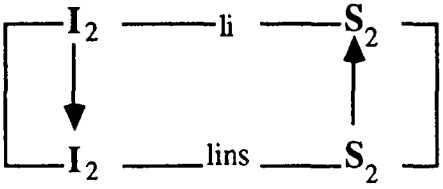
rule 0



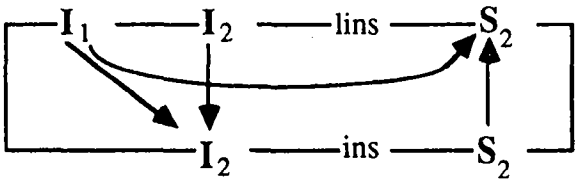
rule 1



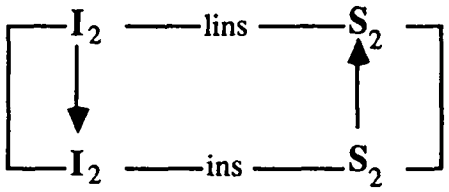
rule 2



rule 3



rule 4



rules 5, 6

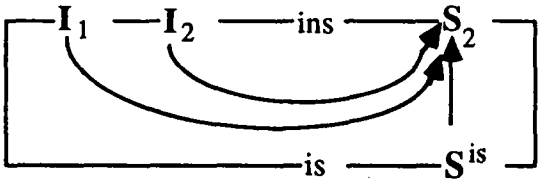


Figure 3 : logical dependency scheme for I_2^x and S_2^x

In rule 3 one needs I_1 to know that N = [lab, _ | L0], thus in term of instructions M - N has the same length as M- L0. Same for S_2. Idem in rules 5 and 6.

From these specifications one can observe that **A1** of x is the address of the first instruction derived from x, as **A2** is the address of the "next" one after x.

(5.4) P4 : adding a label-table :

The label-table will be handled by two arguments added to all the non-terminal symbols but the axiom "prog". They will be denoted **T1** and **T2** and an inherited (synthesized) assertion will specify **T1 (T2)**.

Thus for all x in { li, lins, ins}:

I_3^x : **T1** is the table of the labels corresponding to repr(**x1**).

S_3^x : **T2** is the table of the labels corresponding to repr(**x1**) repr(**x2**).

More formally :

I_3^x : **T1** = l-table(repr(**x1**)) for all x in { li, lins, ins}

S_3^x : **T2** = l-table(repr(**x1**) repr(**x2**)) for all x in { li, lins, ins}

in which it is assumed that l-table (x) is the list of all the pairs [label, N or number] in which all labels are those appearing in x. The second element of the pairs may be unknown (a variable N).

Using two arguments T1 and T2 in state of one only avoids the use of sophisticated functions like the merging of two l-tables. We will use the updating function "test-or-incl".

The program P4 will be given without the arguments A1 and A2 which can be found in P3.

P4

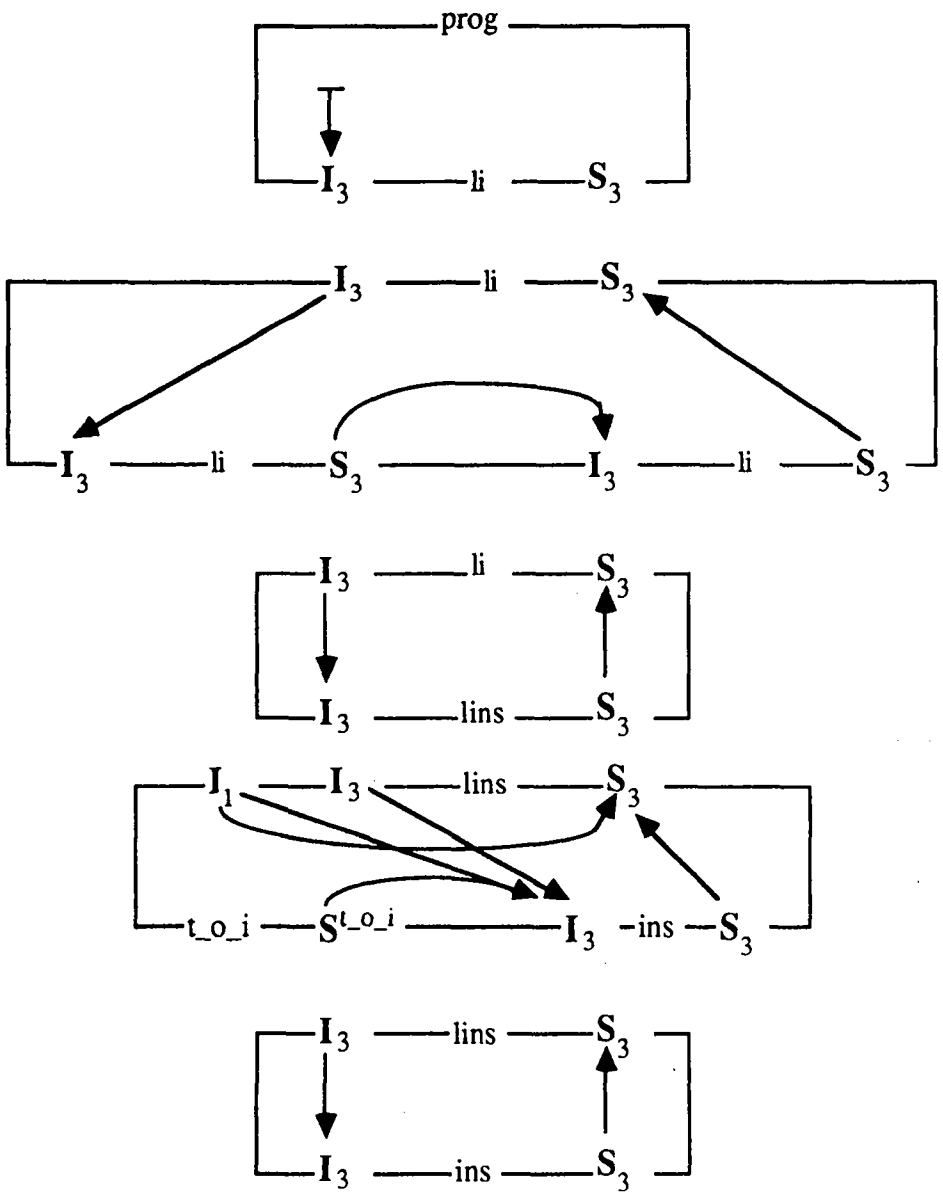
prog(M-L) ← li(M- M, M- L, [], T2)

li(M- L0, L0- L2, T1, T2) ← li(M- L0, L0- L1, T1, T3), li(M- L1, L1- L2, T3, T2)
li(M, L, T1, T2) ← lins(M, L, T1, T2)

lins(M- N, [lab, : | L0]- L1, T1, T2) ← test-or-incl(T1, [lab, A1],T3),
ins(M- L0, L0- L1, T3, T2)
lins(M, L, T1, T2) ← ins(M, L, T1, T2)

ins(M, [ins | L]- L, T, T) ← A2 is A1 + 1
ins(M, [if, expr, thengoto, lab | L]- L, T1, T2)
← A2 is A1 + 1,
test-or-incl(T1, [lab, X], T2)

The proofs are obvious, following the logical dependency schemes of Fig. 4.



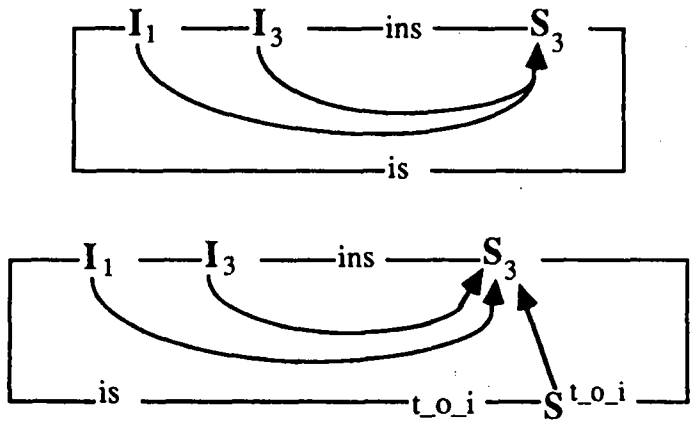


Figure 4 : sound logical dependency scheme for P4

(5.5) P5 : adding the generated code :

In order to avoid the use of concatenation of lists, the generated code will be represented by a difference-list. Thus we will add one argument (denoted **C** in place of **X7**) whose specification is :

S_4^x : repr(C) = trans(repr(**X2**)) for all x in { prog, li, lins, ins}

To simplify the presentation of **C** we do not repeat in **P5** the arguments **X1**, **A1**, **A2**, **T1**, **T2**.

P5

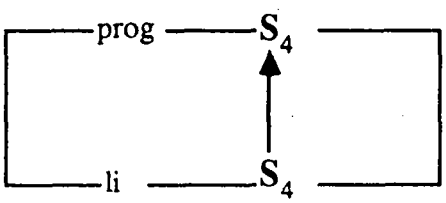
prog(L, C) ← li(L, C)

li(L0- L2, C1- C3) ← li(L0- L1, C1- C2), li(L1- L2, C2- C3)
li(L, C) ← lins(L, C)

lins([lab, : | L0]- L1, C) ← test-or-incl(T1, [lab, A1],T3), ins(L0- L1, C)
lins(L, C) ← ins(L, C)

ins([ins | L]- L, [[A1, ins] | C]- C) ← A2 is A1 + 1
ins([if, expr, thengoto, lab | L]- L, [[A1, expr, X] | C] -C)
← A2 is A1 + 1,
test-or-incl(T1, [lab, X], T2)

The proofs are obvious, following the logical dependency schemes of Fig. 5.



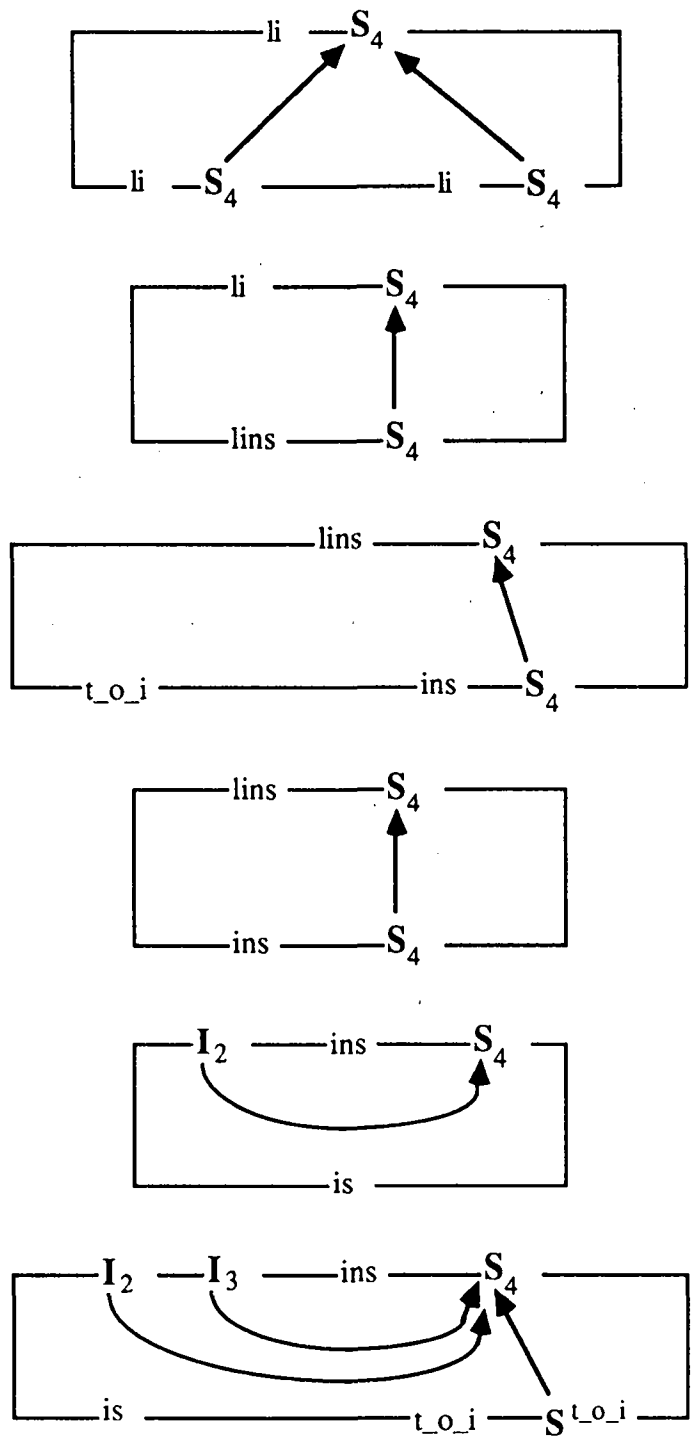


Figure 5 : sound logical dependency scheme for P5

Note that a label in the generated code may remain a variable if it is not defined in the source program and if a label is defined twice (i.e. with different addresses) there will not be object code as there is no possible proof-tree for **P4** in this case.
The proof of soundness of the logical dependency schemes has now been completed. As the variables appearing in the first argument of li, lins and ins are not used in the arguments A1, A2, T1, T2 or C it is possible to remove the first argument for these predicates. Thus the final program is the following :

(5.6) P6 : final program

prog(L, C) ← li(L, 1, A2, [], T2, C)

li(L0- L2, A1, A2, T1, T2, C1- C3) ← li(L0- L1, A1, A3, T1, T3, C1- C2),
li(L1- L2, A3, A2, T3, T2, C2- C3)

li(L, A1, A2, T1, T2, C) ← lins(L, A1, A2, T1, T2, C)

lins([lab, : | L0]- L1, A1, A2, T1, T2, C) ← test-or-incl(T1, [lab, A1],T3),
ins(L0- L1, A1, A2, T3, T2, C)

lins(L, A1, A2, T1, T2, C) ← ins(L, A1, A2, T1, T2, C)

ins([ins | L]- L, A1, A2, T, T, [[A1, ins] | C]- C) ← A2 is add1(A1)

ins([if, expr, thengoto, lab | L]- L, A1, A2, T1, T2, [[A1, expr, X] | C] -C)
← A2 is add1(A1),
test-or-incl(T1, [lab, X], T2)

test-or-incl(T, I, T) ← is-already-in(T, I)

test-or-incl(T, I, [I | T]) ← is-not-already-in(T, I)

is-already-in([I | T], I) ←

is-already-in([I1 | T], I2) ← is-already-in(T, I2)

is-not-already-in([], I) ←

is-not-already-in([I1 | T], I2) ← diffI(I1, I2), is-not-already-in(T, I2)

diffI([E1, A1], [E2, A2]) ← diff(E1, E2)

The annotations are: (**X1** is the removed first argument)

Inherited : for all x in { li, lins, ins}:

I_1^x : ∃ α prog → repr(**X1**) x ∃ and α M, N, L **X1** = M-N and **X2** = N-L

I_2^x : **A1** = length(repr(**X1**))+ 1

I_3^x : **T1** = l-table(repr(**X1**))

Synthesized :

S_1^x : x → repr(**X2**) for all x in { prog, li, lins, ins}

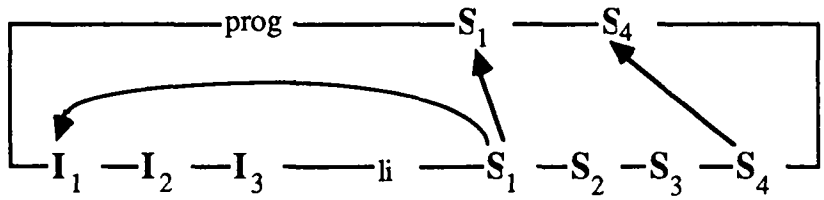
S_2^x : **A2** = length(repr(**X1**) repr(**X2**))+ 1 for all x in { li, lins, ins}

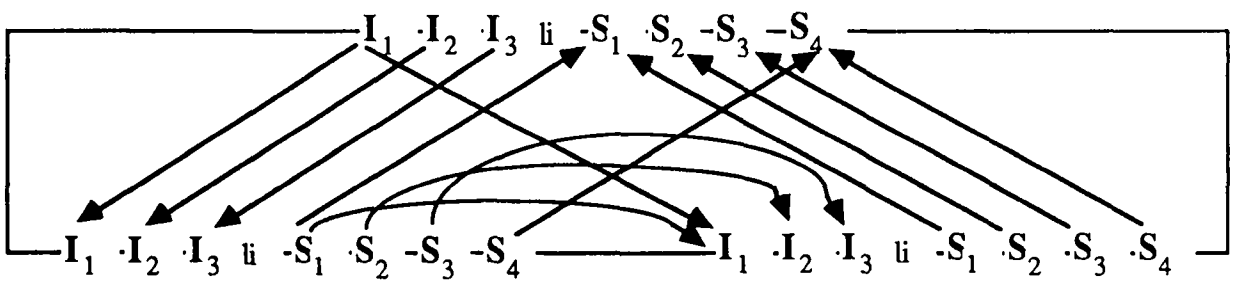
S_3^x : **T2** = l-table(repr(**X1**) repr(**X2**)) for all x in { li, lins, ins}

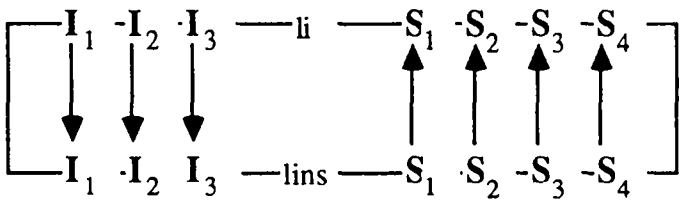
S_4^x : repr(C) = trans(repr(**X2**)) for all x in { prog, li, lins, ins}

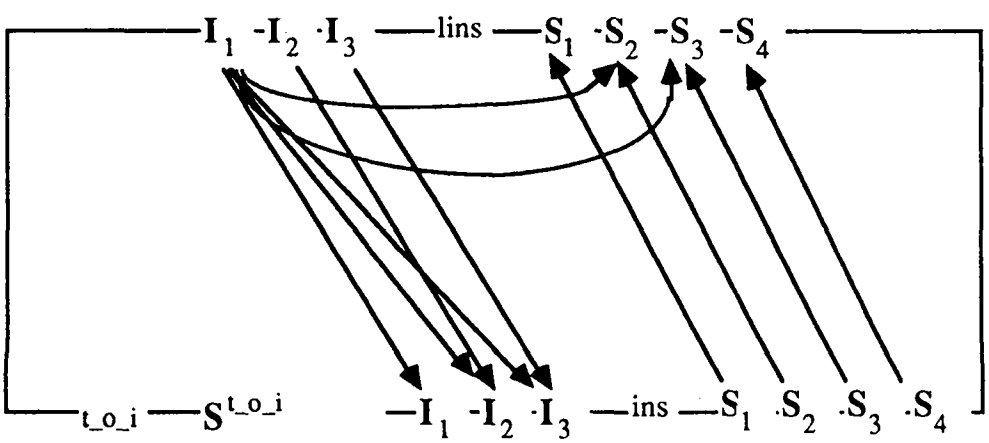
and :
$S^{\text{test-or-incl}}$, $S^{\text{is-already-in}}$, $S^{\text{is-not-already-in}}$, S^{diffI}, S^{diff}, S^{is}

The logical dependency schemes are obtained by merging all the previous partial dependency schemes :









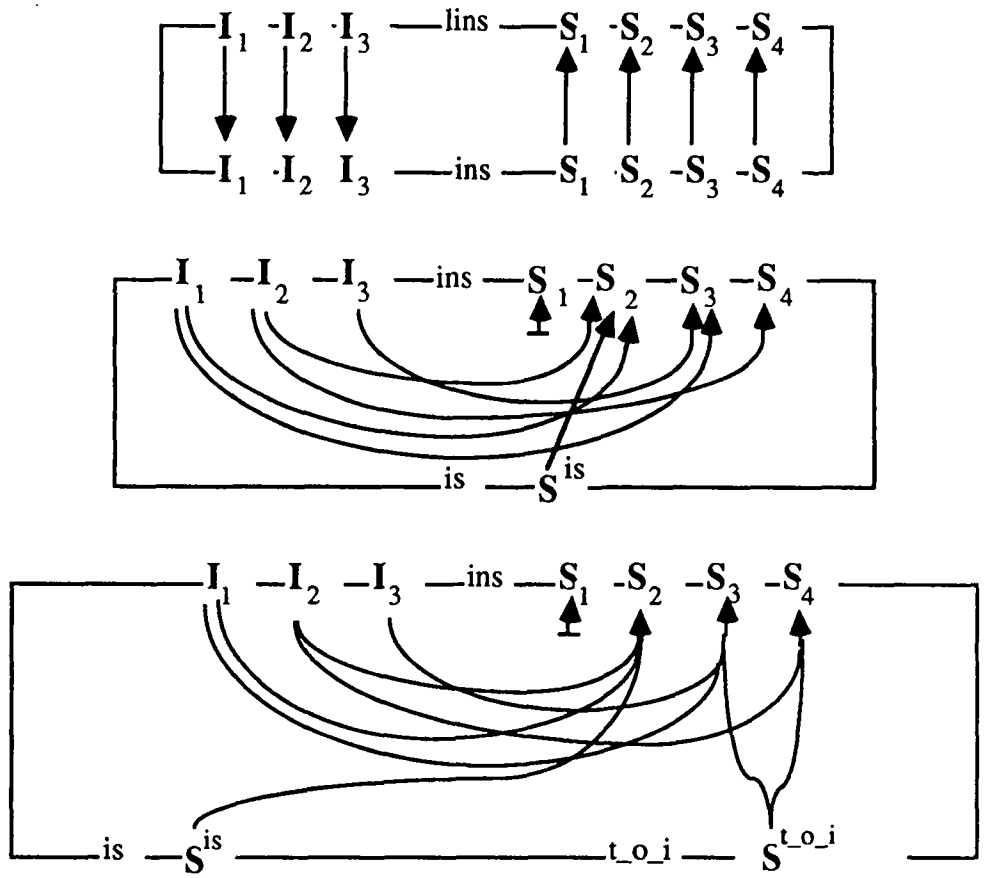


Figure 6

As the logical dependency scheme is sound and non-circular, the annotation is valid, hence every proof-tree of root prog(S, C) satisfies :

S_1^{prog} : prog $\xrightarrow{+}$ repr(S) and S_4^{prog} : repr(C) = trans(repr(S))

Remark that the non-circularity is not trivial here, but the way the annotation has been obtained step by step avoided the production of a circularity as long as the new assertions was non-circular and used the old one's only.

(5.7) The program P6 is well-typed

One uses the following annotation which shows that all arguments of the predicates in a complete proof-tree of root "prog" respect their type and that symbol tables are represented by a list of items whose first elements are different ground atoms and the second is an integer or a variable (all this information is condensed in "list-of-items"). Note that there is no inherited assertion associated to the root "prog". In particular the "atomic" labels may be represented by any kind of ground term.

Inherited assertions

T_1^X : list-of-items (X4) $\wedge$ noduplic (X4), X $\in$ {li, lins, ins}

$T_3^{test\text{-}or\text{-}incl}$: list-of-items (toi1) $\wedge$ item (toi2)

T_4^X : list-of-items (X1) $\wedge$ item (X2), X $\in$ {is-already-in, is-not-already-in}

T_5^{diffI} : item(diffI1) ∧ item(diffI2)

T_6^{diff} : atom(diff1) ∧ atom(diff2)

T_7^X : integer(X2), X ∈ {li, lins, ins, is}

Synthesized assertions

U_1^X : is-a-d_list (X1), X ∈ {prog, li, lins, ins}

U_2^{prog} (L, C) : is-a-d_list (C) ∧ well-typed (repr(C))

U_2^X : is-a-d_list (X6) ∧ well-typed (repr(C)), X ∈ {li, lins, ins}

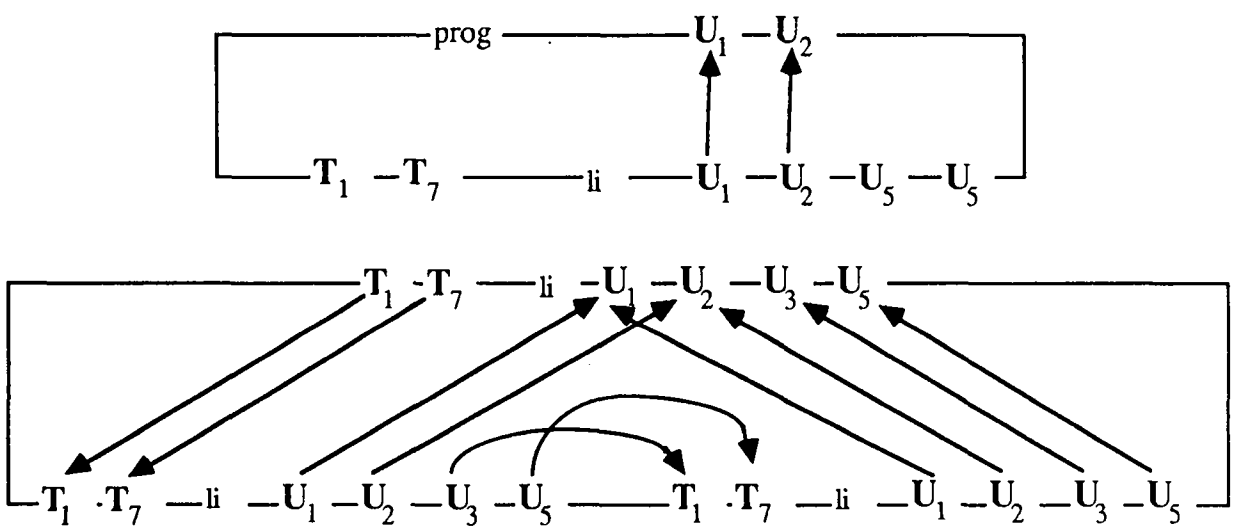
U_3^X : list-of-items (li5) ∧ noduplic (li5), X ∈ {li, lins, ins}

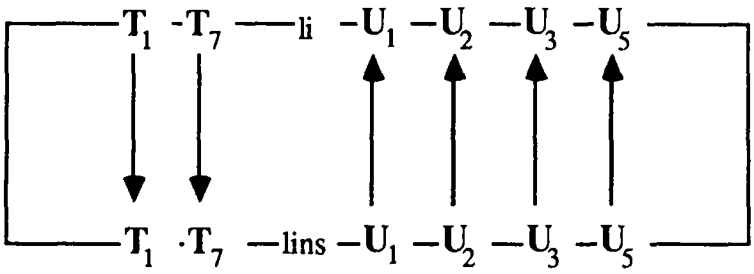
$U_3^{test\text{-}or\text{-}incl}$: list-of-items (toi3)

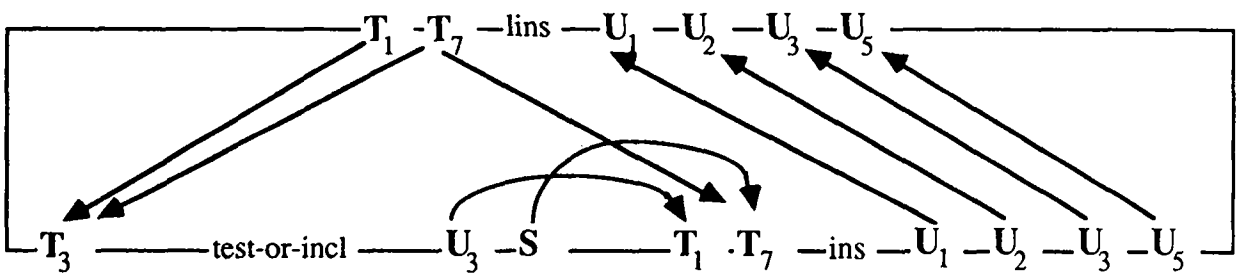
U_5^X : integer(X3), X ∈ {li, lins, ins, is}

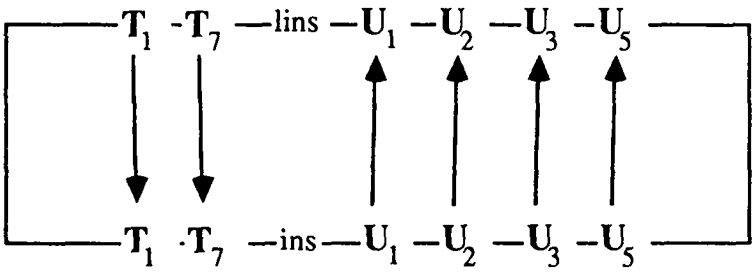
Two assertions need some comments (T_0, U_2^{prog}). In T_0 one requires "well-typed(repr(L))" ; as L is a difference list by assertion U_1^{prog} the "well-typed" predicate means only that the labels of the source program must be ground atoms (needed to satisfy "list-of-items", "item" and "atom" properties). Furthermore in U_2^{prog} "well-typed (repr(C))" means that the generated code C is a list of pairs or triples whose components have the right type described at the beginning of section 5 (in particular the first element is an integer).

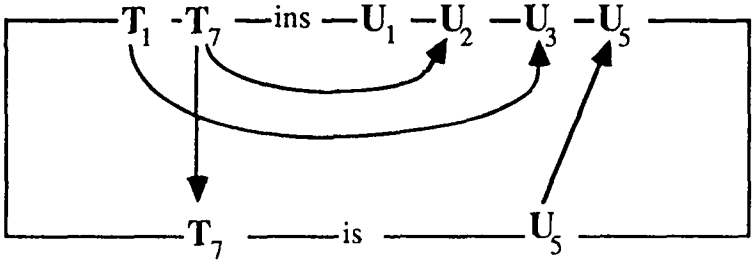
The proofs are obvious. We just draw the local dependency scheme (Fig. 7), most of them are obvious. Notice that some of them are just inherited.

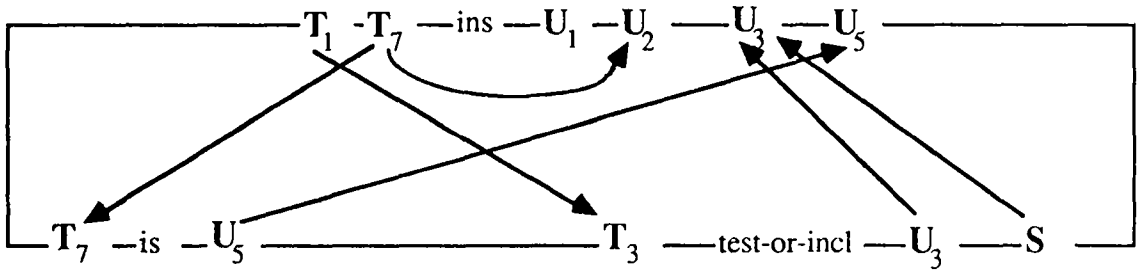


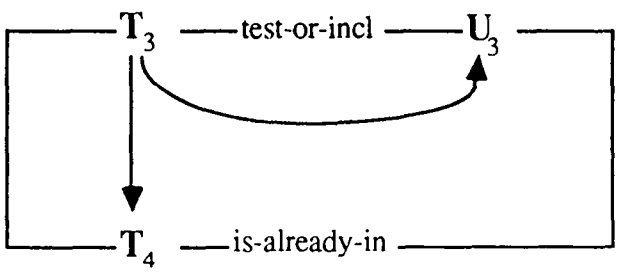

T1 -T7 —li -U1 —U2 —U3 -U5
T1 ·T7 —lins —U1 —U2 —U3 —U5


T1 -T7 —lins —U1 —U2 —U3 —U5
T3 —test-or-incl —U3 —S —T1 .T7 —ins —U1 —U2 —U3 —U5


T1 -T7 —lins —U1 —U2 —U3 —U5
T1 ·T7 —ins —U1 —U2 —U3 —U5


T1 -T7 —ins —U1 —U2 —U3 —U5
T7 —is —U5


T1 -T7 —ins —U1 —U2 —U3 —U5
T7 —is —U5 —T3 —test-or-incl —U3 —S


T3 —test-or-incl —U3
T4 —is-already-in

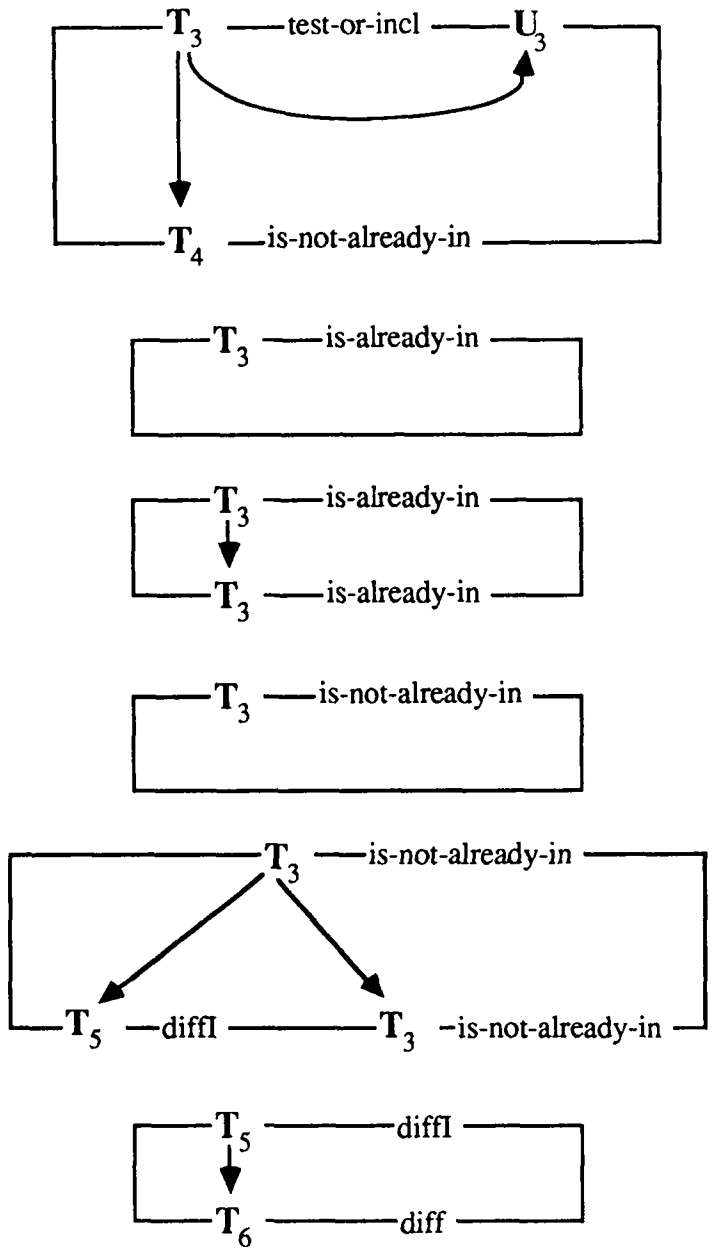


Figure 7 : sound logical dependency scheme for proving the well-typedness of P6

6. Comparison with other works on partial correctness in Logic Programming

The first presentation of the inductive proof method used to establish the computational validity of a specification appears in [Cla 79] mainly based on the axiomatic view of sections 3.15 and 3.16. When induction in the theories defined by what is called here Ax (axioms defining the "input relations" or in some sense the pre-interpretations) it is called **structural induction** (on the "input relations") in the sense of Burstall and Darlington [BD 75].

When a formula is proven using statment4 in (3.16), i.e. using an inductive specification in DEN(P), then it is called **computational induction** in the sense of Manna [Man 74].

Computational induction, as structural induction can be used to prove general properties about the program, holding in DEN(P). Clark remarks also that the "pure" inductive assertion method, called fixpoint induction by Park [Pa 69], is a sound rule to prove the computational validity. He calls this rule the "Consequence Verification Method".

In most of his examples Clark considers that the set of axioms Ax is defined by a logic program. Hence statment 5 (section 3.16) is extensively used (Ax is the completion of a logic program).

We adopted the same point of view in section (3.16) showing that these different kinds of inductions are the same if one considers that the axioms Ax are definite clauses also.

The first presentation of the proof method by annotations appears in [CD 88] where it is restricted to the use of assertions to prove valid specifications. We have given here a slightly more general presentation of the method (adapted to the case of logic programming) : we use the annotation method to prove properties holding inside the proof-trees, not only at the root. With some adaptation the method can be used to prove dynamic properties.

For example in [DrM 87] a scheme analogous to a L-LDS is used to prove run-time properties, but if ground proof-trees are considered the kind of used assertions can be viewed as assertions holding in the proof-trees, hence in particular at the root.

In Bossi and Cocco [BC 89] an annotation based on a L-LDS is used to prove partial correctness of "modules" w.r.t. a specification (computational validity). They use the axiomatic view and we have shown that the restriction to L-LDS is not necessary.

In Hogger [Hog 81, 84] it is assumed that the axioms Ax are a DCP which includes an (other) definition of the predicates in PRED.The definition of "inductive for P" becomes just : $S \models P$, i.e. the clauses of P are theorems in Ax = S. Theorem (3.15.3) is a reformulation of the sufficient criterion given by Hogger, in this particular case, when Ax contains an other axiomatisation of P also.

In Shapiro and Sterling's book [SS 86] the inductive proof method is used to validate specifications (computational validity) expressed on Herbrand interpretations, but the method itself is not investigated.

Kanamori [Kan 86, KS 86] developed extensively the idea that execution of a DCP can be used to prove that general formulas are logical consequences of COMP(P) or valid in DEN(P). The work is restricted to DCP's and to formulas called S-formulas (of the form $\forall \bar{x} \exists \bar{y} F(\bar{x} \bar{y})$). In [Kan 86] the "extended execution deduction rule", which is an extension of the SLDNF resolution, is defined and proved complete ; i.e. given a DCP P and a S-formula F, $COMP(P) \models F$ if and only if starting with

F, the extended execution deduction rules applied to P permit to derive the "true" goal. This deduction rule is implemented in the system ARGUS [KFHM 86] with the induction proof method which is used to prove S-formulas valid in DEN(P). These ideas have given rise to many improvments [Fri 88] and analogous ideas have been studied as in [HS 84].

In Deville [Dev 88] and Deransart and Ferrand [DF 87, 89a, b] the extension of the inductive proof method for logic programs with negations (normal programs) is presented. For DCP's the completion COMP(P) has a unique least term models ; for normal programs it is not anymore true and the completion may have many uncomparable minimal term models. Deville assumes that there exists a unique term model. This property is undecidable in general but is verified by construction of the (normal) program. His method needs to know the strongest specification and is introduced rather with the purpose to build correct programs rather than to prove them corrects afterwards. Deransart and Ferrand have basically the same approach (the specification, if expressed in a term model, is a well-founded model (in a sense which is out of the scope of this paper) of COMP(P)), but they give a modular version of the proof method for partial correctness which reduces to the inductive proof method in case of a DCP, and is suitable for weaker specifications than the strongest one. However the proofs of weaker (or partial) specifications need stronger (or larger) specifications of completeness.

Conclusion :

In this paper, we have investigated the problem of proving the partial correctness of a logic program — restricted to definite clauses and some possible extensions — w.r.t. a specification in an unified framework. We have considered two cases : "informal" proofs (i.e. performed in some "known" domain) or "formal" proofs (i.e. performed in some axiomatized domains), and we have introduced two proof methods : the inductive method and the annotations method, the first one beeing a particular case of the second one.

We have introduced two notions of validity with the purpose to obtain completeness results of the methods : "validity" and "computational validity". From the point of view of a (logic programming) programmer the second notion is the only interesting one.

Different notions of "completeness" have been used :

1) completeness of a set of axioms.

   Obviously when the axiomatic view is used – and automatized proofs are performed – if the set of axioms is not complete (i.e. if the theory is not decidable) it does not make any sense to speak about other concepts of completeness : any method will remain incomplete.

2) completeness of the method (inductive assertions, or annotations) expressed by a necessary condition for validity. Neither with the computational validity nor in general if the axiomatic view is taken, even with a decidable theory, the completeness can be reached. If so there is still a problem.

3) relative completeness : completeness of the method can be reached only if the specification langage is powerful enough. We have shown that if one restricts the specification language to the first order logic the completeness of the method does not hold in general, even for pure DCP's.

The presented proof methods can be viewed as new deduction rules and we have shown that they can be used for more general purposes than just to prove the validity of specifications : proof of validity of general formulas in one model (DEN) or proof of properties inside the proof-trees. The

annotation method, whose adaptation to the field of logic programming is new, seems to be particularly well suited for this latter purpose.

We have compared this presentation of the proof methods with other known works and we have shown that all of them use with a slightly different formalism, or different conditons, the same method : the inductive — or fixpoint — one or in the best case the annotation method with a L-LDS.

We think that the existence of simple and powerful proof methods of validity is one of the caracteristics of logic programming which make it attractive. We did this extensive study because of two main reasons :

1) They are very general (complete) and simple (especially if a short inductive assertion is proven). As such they can be taught together with a PROLOG dialect and may help the user to detect useful properties of the written axioms. In the case of large programs the second method may help to simplify the presentation of a proof using shorter assertions and clear logical dependencies between assertions.

2) Valid specifications are the basic elements used in the proofs of all other desirable logic program properties as completeness, "run-time" properties, termination such as shown in [DF 88] or safe use of the negation [Llo 87]. For example any proof of termination with regards to some kind of used goals and some strategy will suppose that, following the given strategy, some sub-proof-tree has been successfully constructed and thus that some previously chosen atoms in the body of a clause satisfy their specifications. Thus correctness proofs appear to be a way of making modular proofs of other properties also. In fact the validity of a specification can be established independently of any other property.

As most of the other manipulations on logic programs (proof of other properties, program transformations ...) use valid specifications, one may except that the same results of (in)completeness hold also in these methods. This is fortunately not a reason not to try to validate a program and to develop good methods well adapted to the kind of property one wants to verify.

Acknowledgments

We are indebted to B. COURCELLE with whom most of the basic ideas have been developed in the field of attribute grammars and to G. FERRAND and J.P. DELAHAYE who suggested some questions and helped to clarify this text.

Annex

Missing proofs of section 3.15.

Claim : the only - if part of the theorem 3.15.3 does not hold.

By hypothesis for every $\mathbb{D}$ model of Ax, if $p(\bar{v}) \in PTR_{\mathbb{D}}(P)$ then $\mathbb{D} \models S^p[\bar{v}]$. Let us consider some $\mathbb{D}$ and $S = S_{P, \mathbb{D}}$; $Ax \models P\,[S_{P, \mathbb{D}}]$ does not hold necessarily in all models of Ax (it does for $\mathbb{D}$ by hypothesis).

Claim : the only-if part of the proposition 3.15.4 does not hold.

By the same reasonning one gets : $Ax \cup IFF(P) \models p(\bar{x}) \Rightarrow S^p{}_{P, \mathbb{D}}(\bar{x})$. This holds by hypothesis in all $\mathbb{D}$-based models of IFF(P) but not necessarily in all models of $Ax \cup IFF(P)$.

Claim : both proof methods of section 3.15 are incomparable.

Assume that both properties hold simultaneously :

(1) $Ax \cup IFF(P) \models p(\bar{x}) \Rightarrow S^p[\bar{x}]$.for all p in PRED

(2) There exists S' such that

a) $Ax \models P[S']$

b) $Ax \models S' \Rightarrow S$

Given a model $\mathbb{D}$ of Ax and $\mathbb{D}_P$ ($\mathbb{D}$ extended with $PTR_{\mathbb{D}}(P)$), let us consider that S is $S_{P, \mathbb{D}}$, then one gets $\mathbb{D}_P \models S_{P, \mathbb{D}} \Leftrightarrow S'$ ($S_{P, \mathbb{D}} \Rightarrow S'$ by 2-a and $S' \Rightarrow S_{P, \mathbb{D}}$ by 2-b). Hence all the formulas $S_{P, \mathbb{D}}$ are equivalent to a unique formula S' in all models of Ax, i.e. the strongest specification for P corresponds to the same formula in all models of Ax. This is not true in general. (It is in particular if Ax is a DCP and one considers term models only).

References

[Apt 87] K. R. Apt : introduction to Logic Programming. University of Texas at Austin, TR 87-35, September 1987.

[AvE 82] K.R. Apt, M.H. Van Emden : Contributions to the theory of Logic Programming. JACM V29, N° 3, pp 841-862, July 1982.

[Bal 78] K. Balogh : On an Interactive Program Verifier of PROLOG Programs. Colloquia Math. societatis, Janos, Bolyai 26, MLCS, Saly'otzyan, Hungary 1978.

[Bau 88] M. Baudinet : Proving Termination Properties of PROLOG Programs : A semantic Approach. Report STAN-CS 88-1202, March 1988.

[BC 89] A. Bossi, N. Cocco : Verifying Correctness of Logic Programs, Tapsoft'89, Barcelone. LNCS 352.

[BD 75] R. M. Burstall, J. Darlington : Some Transformations for Developping Recursive Programs. Proc. Int. Conf. on Reliable Software. Los Angeles, 1975.

[BM 88] R. Barbutti, M. Martelli : A tool to check the Non-Floundering of Logic Programs and goals. PLILP'88. lLNCS 348, May 1988.

[Bru 87] M. Bruynooghe & al : Abstract Interpretation : the Global Optimization of PROLOG programs. SLP'87, San Francisco, 1987.

[CD 88] B. Courcelle, P. Deransart : Proof of partial Correctness for Attribute Grammars with application to Recursive Procedure and Logic Programming. Information and Computation 78, 1, July 1988 (First publication INRIA RR 322 - July 1984).

[CL 88] H. Comon, P. Lescanne : Equational Problems and Disunification. INRIA RR 904, September 1988.

[Cla 79] K.L. Clark : Predicate Logic as a Computational Formalism. Res. Mon. 79/59 TOC. Imperial College, December 1979.

[Coo 78] S.A. Cook : Soundness and Completeness of an Axiom System for Programs Verification. SIAM Journal. Comput. V7, n° 1, February 1978.

[Cou 84] B. Courcelle : Attribute Grammars : Definitions, Analysis of Dependencies, Proof Methods. In Methods and Tools for Compiler Construction, CEC-INRIA Course (B. Lorho ed.). Cambridge University Press 1984.

[CT 77] K. L. Clark, S. A. Tärlund : A first Order Theory on Data and Programs. IFIP'77, North Holland, 1977.

[Der 83] P. Deransart : Logical Attribute Grammars. Information Processing 83, pp 463-469, R.E.A. Mason ed. North Holland, 1983.

[Der 89] P. Deransart : Proofs of Declarative Properties of Logic Programs. Tapsoft'89, Barcelona. LNCS 352. March 89.

[Dev 87] Y. Deville : A Methodology for Logic Program Construction.PhD Thesis, Institut d'Informatique, Facultés Universitaires de Namur (Belgique), February 1987.

[Dev 88] Y. Deville : A correctness Definition for Logic Programming. Institut d'Informatique. Namur University (Belgium). RP 88/8, 1988.

[DF 87] P. Deransart, G. Ferrand : An Operational Formal Description of PROLOG. 4th Symposium on Logic Programming. August 31-September 4, 1987. SLP'87, San Francisco.

[DF 88] P. Deransart, G. Ferrand : Logic Programming, Methodology and Teaching. K. Fuchi, L. Kott editors, French Japan Symposium, North Holland, pp 133-147, August 1988.

[DF 89a] P. Deransart, G. Ferrand : Methodological View of Logic Programming with Negation. INRIA RR 1011. April 1989.

[DF 89b] P. Deransart, G. Ferrand : Proofs Methods and Declarative Diagnosis in Logic Programming. Tutorial ICLP'89, Lisbon, June 89.

[DJL 88] P. Deransart, M. Jourdan, B. Lorho : Attribute Grammars : Definitions, Systems and Bibliography, LNCS 323, Springer Verlag, August 1988.

[DM 85] P. Deransart, J. Maluszynski : Relating Logic Programs and Attribute Grammars. J. of Logic Programming 1985, 2, pp 119-155. INRIA, RR 393, April 1985.

[DM 89] P. Deransart, J. Maluszynski : A Grammatical View of Logic Programming. PLILP'88, Orléans, France, May 16-18, 1988, LNCS 348, Springer Verlag, 1989.

[DrM 88] W. Drabent J. Maluszynski : Inductive Assertion Method for Logic Programs. TCS 59, July 1-2, 1988.

[Fer 85] G. Ferrand : Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Methods. J. of Logic Programming Vol. 4, pp 177-198, 1987.

[FGK 85] N. Francez, O. Grumberg, S. Katz, A. Pnuelli : Proving Termination of Prolog Programs. In "Logics of Programs, 1985", R. Parikh Ed., LNCS 193, pp 89-105, 1985.

[FK 86] H. Fujita, T. Kanamori : Formulation Induction Formulas in Verification of Prolog Programs. TR 097, ICOT, 1984, (8^{th} Conf. on Automated Deduction 1986).

[Fri 88] L. Fribourg : Equivalence-Preserving Transformations of Inductive Properties of Prolog Programs. ICLP'88, Seattle, August 1988.

[Hog 81] C.J. Hogger : Derivation of Logic Programs. JACM 28, 2, April 1981.

[Hog 84] C.J. Hogger : Introduction to Logic Programming. APIC Studies in Data Processing n° 21, Academic Press, 1984.

[HS 84] J. Hsiang, M. Srivas : On proving First Order Inductive Properties in Horn Clauses. DSC, State University of NY at Stony Brook. TR 84/075, April 84.

[Kan 86] T. Kanamori : Soundness and Completeness of Extended Execution of Proving Properties of Prolog Programs. ICOT, TR 175, May 1986.

[KFHM 86] T. Kanamori, H. Fujita, K. Horiuchi, M. Maeji : ARGUS/V : a System for Verification of Prolog Programs. ICOT TR-176, 11 p., May 1986.

[Llo 87] J. W. Lloyd : Foundations of Logic Programming. Springer Verlag, Berlin, 1987.

[Man 74] Z. Manna : Mathematical Theory of Computation. Mc Graw Hill, NY, 1974.

[Pad 88] P. Padawitz : Inductive Proofs of Constructor-based Horn Clauses. Universität Passau, Report MIP-8810.

[Par 69] D. Park : Fixpoint Inductin and Proofs of Program Properties. Machine Intelligence 5, pp 59-76, 1969.

[PV 86] J. Potter, T. Vasak : Characterisation of Terminating Logic Programs. SLP'86, Salt Lake City, 1986.

[SS 86] L. Sterling, E. Y. Shapiro : The Art of Prolog. MIT Press, 1986.

[Tar 55] A. Tarski : A Lattice-theoretical fixpoint theorem and its Applications. Pacific J. of Maths, 5, pp 285-309, 1955.

[Tär 86] S. A. Tärlund : Logic-programming from a Logic Point of View. SLP'88, Salt Lake City 1986.

[Vas 86] T. Vasak : Toward a Methodology for Logic Programming. Ph. D dissertation, DCS, University of New South Wales, April 86.

[Wan 78] M. Wand : A new Incompleteness result for Hoare's System. JACM 25, pp 168-175, 1978.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399